""" Python """


```
// Printing
print("Hello")
 // Error // Unexpected indent
 print("World")
print(!)

// Variables
'''
Names are case-sensitive may begin with:
   letters, $ , _
After, may include
   letters, numbers, $, _
Convention says
   Start with lowercase word, then additional words are separated
   by underscores
   ex. my_first_vriables
'''
// Multi-line comment
''' '''
// String
name = "Mike"
// Integer
age = 30
// Decimal
gpa = 3.5
// Boolean -> True/False
is_tall = True
// Modify variable instead of Mike
name = "John"
// Concatenating
print("Your name is " + name)
print("Your name is", name)

// Casting & Converting
print( int(3.14))
print( float(3))
print( str(True))
print( int("50") + int("70"))

// Strings
greeting ="Hello"
#indexes:  01234
print( len(greeting))
print( greeting[0])
print( greeting[-1])
print( greeting.find("llo") )
print( greeting.find("z") )
print( greeting[2:] )
print( greeting[2:3] )

// Numbers
// Basic Arithmetic: +, -, /, *
print( 2 * 3 )
// Basic Arithmetic: , -, /, *
```

```
print( 2**3 )
// Modulus Op.: returns remainder of 10/3
print( 10 % 3 )
// Order of operations
print( (1 + 2) * 3 )
// int's and doubles
print(10 / 3.0)
num = 10
// +=, -=, /=, *=
num += 100
print(num)
++num
print(num)
// Math class has useful math methods
import math
print( pow(2,3) )
print( math.sqrt(144) )
print( round(2.7) )

// User Input
name = input("Enter your name: ")
print("Hello", name + "!")

num1 = int(input("Enter First Num: "))
num2 = int(input("Enter Second Num: ")
print(num1+num2)

// Lists (similar to arrays from other languages)
lucky_numbers =[4, 8, "fifteen, 16, 23, 42.0"]
      //indexes   0  1      2       3   4     5
lucky_numbers[0] = 90
print(lucky_numbers[0])
print(lucky_numbers[1])
print(lucky_numbers[-1])
print(lucky_numbers[2:])
print(lucky_numbers[2:4])
print(len(lucky_numbers))

// N dimensional lists
numberGrid = [ [1,2], [3,4] ]
numberGrid[0][1] = 99
print(numberGrid[0][0])
print(numberGrid[0][1])

// List functions
friends = [ ]
// This will append at the end of the list
friends.append("Oscar")
friends.append("Angela")
// Will add at the position that we want
friends.insert(1, "Kevin")
friends.remove("Kevin")
print( friends )
print( friends.index("Oscar") )
print( friends.count("Angela") )
friends.sort()
print( friends )
friends.clear()
print( friends )

// Tuples
lucky_numbers = (4, 8, "fifteen", 16, 23, 42.0)
```

```
    //indexes    0 1    2       3  4    5
// Error // Tuples cannot be modified
lucky_numbers[0] = 90
print(lucky_numbers[0])
print(lucky_numbers[1])
print(lucky_numbers[-1])
print(lucky_numbers[2:])
print(lucky_numbers[2:4])
print(len(lucky_numbers))

// Functions (re-use block of code)
// num1 is a parameter, we can give num2 a default value of the user does not give value for parameter 2
def add_numbers(num1, num2=99):
  return num1 + num2
sum = add_numbers(4, 3)
print(sum)

// If statements (conditionals)
is_student = False
is_smart = False
// Use can use "and", "or" logic OR negation operator with "not(condition)"
if is_student and is_smart:
  print("You are a student")
elif is_student and not(is_smart):
  print("You are not a smart student")
else:
  print("You are not a student and not smart")

// >, <, >=, <=, !=, == # you can use comparisons
// Number comparison
if 1 > 3:
  print("number comparison was true")
// String comparison
if "dog" == "cat":
  print("string comparison was true")

// Dictionaries
test_grades = {
  "Andy" : "B+",
  "Stanley" : "C",
  "Ryan" : "A",
  3 : 95.2
}
print( test_grades["Andy"] )
// If the key is not found like Doug in our case then it will default to "No Student Found"
print( test_grades.get("Doug", "No Student Found") )
// Attention bellow is the key 3 not index 3
print( test_grades[3] )

// While loops
index = 1
while index <= 5:
  print(index)
  // Without adding below, our while loop will become infinite loop
  index += 1

// For loops
for index in range(5):
  print(index)
lucky_nums = [4, 8, 15, 16, 23, 42]
for lucky_num in lucky_nums:
  print(lucky_num)
```

```
for letter in "Giraffe":
   print(letter)

// Exception Catching
answer = 10 / int(input("Enter Number: "))
try:
   answer = 10 / int(input("Enter Number: "))
except:
   print("Error")
try:
   answer = 10 / int(input("Enter Number: "))
except ZeroDivisionError as e:
   print(e)
except:
   // Big no
   print("Caught any exception")

// OOP
class Book:
   // Constructor, this is a method when it is going to be called when we create an instance of the book class
   // self refers to the instance of the class
   // self will refer to the object that is going to be created
   // Any function inside class we will pass self inside
   // You always need to use self keyword and that will refer to the object that will be created
   def __init__(self,title,author):
      #title and author are 2 arguments taken
      self.title = title
      self.title = author

   def read_book(self):
      print("Reading", self.title, "by", self.author)

// Creating an object
// We will pass title and author only as self will be passed in automatically
book1 = Book("Harry Potter", "JK Rowling");
// You can modify also title
book1.title = "Half-Blood Prince"
print(book1.title)
book1.read_book()

// Getters & Setters
//It is a design pattern that can be used in OOP that allows you to control the access for the outside code has to the attributes of the class
class Book:
   // Constructor
   // So now by creating all below, anytime I will try to use that title it will automatically go through below functions, so it will not set the attribute directly
   def __init__(self, title, author):
      self.title = title;
      self.author

   // Create the getter (the underscore indicate that this attribute is going to be private)
   @property
   def title(self):
      print("getting title")
      return self._title

   // Create setter
   @title.setter
   def title(self, value):
      print("setting title")
      self._title = value

   @title.deleter
```

```
    def title(self):
        del self._title

    def read_book(self):
        print("Reading", self.title, "by", self.author)

book1 = Book("Harry Potter", "JK Rowling");
book1.title = "Half-Blood Prince"

print(book1.title)
book1.read_book()

// Inheritance
class Chef:
    // So the chef can do 3 things
    def make_chicken(self):
        print("The chef makes chicken")
    def make_salad(self):
        print("The chef makes salad")
    def make_special_dish(self):
        print("The chef makes bbq ribs")
// So Italian chef will inherit functionality from the chef class
class ItalianChef(Chef):
    def make_pasta(self):
        print("The chef makes pasta")
    def make_special_dish(self):
        // It is the same method as the chef class method but we will override it
        print("The chef makes chicken parm")
// We are creating a Chef object
myChef = Chef()
myChef.make_chicken()
myChef.make_special_dish()
// We are creating an Italian Chef object
myItalianChef = ItalianChef()
myItalianChef.make_chicken()
myItalianChef.make_special_dish()
// Using a constructor in a subclass
class Chef:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def make_chicken(self):
        print("The chef makes chicken")
    def make_salad(self):
        print("The chef makes salad")
    def make_special_dish(self):
        print("The chef makes bbq ribs")
class ItalianChef(Chef):
    def __init__(self, name, age, country_of_origin):
        self.country_of_origin = country_of_origin
        // We are calling the super class constructor
        // So we will initalize the name and the age that we inherited from the super class
        // In a way we are overriding the super constructor
        super().__init__(name,age)
    def make_pasta(self):
        print("The chef makes pasta")
    def make_special_dish(self):
        print("The chef makes chicken parm")
myChef = Chef("Gordon Ramsay", 50)
myChef.make_chicken()
myItalianChef = ItalianChef("Massimo Bottura", 55, "Italy")
print(myItalianChef.age)
```

OVERVIEW

Python Syntax

- Python is an example of a very commonly-used modern programming language.
  - C was first released in 1972, Python in 1991.
- Python is an excellent and versatile language choice for making complex C operations much simpler.
  - String manipulation
  - Networking
- Fortunately, Python is heavily inspired by C (its primary interpreter, *Cpython*, is actually written in C) and so the syntax should be a shallow learning curve.

Python Syntax

- To start writing Python, open up a file with the .py file extension.
- Unlike a C program, which typically has to be compiled before you can run it, a Python program can be run without explicitly compiling it first.
- Important note: In CS50, we teach **Python 3**. (Not Python 2, which is also still fairly popular.)

Python Syntax

- **Variables**
- Python variables have two big differences from C.
  - No type specifier.
  - Declared by initialization only.
  - Python statements needn't end with semicolons!

```
x = 54
```

Python Syntax

- **Variables**
- Python variables have two big differences from C.
  - No type specifier.
  - Declared by initialization only.

```
phrase = 'This is CS50'
```

Python Syntax

- **Conditionals**
- All of the old favorites from C are still available for you to use, but they look a little bit different now.

```
if y < 43 or z == 15:
    # code goes here
```

Python Syntax

- **Conditionals**
- All of the old favorites from C are still available for you to use.

```
if y < 43 and z == 15:
    # code block 1
else:
    # code block 2
```

Python Syntax

- **Conditionals**
- All of the old favorites from C are still available for you to use.

```
if coursenum == 50:
    # code block 1
elif not coursenum == 51:
    # code block 2
```

Python Syntax

- **Conditionals**
- All of the old favorites from C are still available for you to use.

```
letters_only = True if input().isalpha() else False
```

```
a = 1
b = 2

letters_only  = True if a == b else False
print(letters_only)
```

Python Syntax

- **Loops**
- Two varieties: `while` and `for`

```
counter = 0
while counter < 100:
    print(counter)
    counter += 1
```

Python Syntax

- **Loops**
- Two varieties: `while` and `for`

```
for x in range(100):
    print(x)
```

Python Syntax

- **Loops**
- Two varieties: `while` and `for`

```
for x in range(0, 100, 2):
    print(x)
```

Python Syntax

- ~~Arrays~~ **Lists**
- Here's where things really start to get a lot better than C.
- Python arrays (more appropriately known as ***lists***) are **not** fixed in size; they can grow or shrink as needed, and you can always tack extra elements onto your array and splice things in and out easily.

Python Syntax

- **Lists**
- Declaring a list is pretty straightforward.

```
nums = []
```

Python Syntax

- **Lists**
- Declaring a list is pretty straightforward.

```
nums = [1, 2, 3, 4]
```

Python Syntax

- **Lists**
- Declaring a list is pretty straightforward.

```
nums = [x for x in range(500)]
```

Python Syntax

- **Lists**
- Declaring a list is pretty straightforward.

```
nums = list()
```

Python Syntax

- **Lists**
- Tacking on to an existing list can be done a few ways:

```
nums = [1, 2, 3, 4]
nums.append(5)
```

Python Syntax

- **Lists**
- Tacking on to an existing list can be done a few ways:

```
nums = [1, 2, 3, 4]
nums.insert(4, 5)
```

Python Syntax

- **Lists**
- Tacking on to an existing list can be done a few ways:

```
nums = [1, 2, 3, 4]
nums[len(nums):] = [5]
```

Python Syntax

- **Tuples**
- Python also has a data type that is not quite like anything comparable to C, a *tuple*.
- Tuples are ordered, immutable sets of data; they are great for associating collections of data, sort of like a struct in C, but where those values are unlikely to change.

Python Syntax

- **Tuples**
- Here is a list of tuples:

```
presidents = [
    ("George Washington", 1789),
    ("John Adams", 1797),
    ("Thomas Jefferson", 1801),
    ("James Madison", 1809)
]
```

Python Syntax

- **Tuples**
- This list is iterable as well:

```
for prez, year in presidents:
    print("In {1}, {0} took office".format(prez, year))
```


```
presidents = [
    ("George Washington", 1789),
    ("John Adams", 1797),
    ("Thomas Jefferson", 1801),
    ("James Madison", 1809)
]
```


Python Syntax

- **Tuples**
- This list is iterable as well:

```
for prez, year in presidents:
    print("In {1}, {0} took office".format(prez, year))
```


```
presidents = [
    ("George Washington", 1789),
    ("John Adams", 1797),
    ("Thomas Jefferson", 1801),
    ("James Madison", 1809)
]
```


```
In 1789, George Washington took office
In 1797, John Adams took office
In 1801, Thomas Jefferson took office
In 1809, James Madison took office
```

Python Syntax

- **Dictionaries**
- Python also has built in support for **dictionaries**, allowing you to specify list indices with words or phrases (*keys*), instead of integers, which you were restricted to in C.

Python Syntax

- **Dictionaries**

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}
```

Python Syntax

- **Dictionaries**

```
pizzas["cheese"] = 8

if pizza["vegetables"] < 12:
    # do something

pizzas["bacon"] = 14
```

```
list = {
    "apple":12,
    "orange":15
    }

if list["apple"] <= 12:
    print("you have less than 12 apples")
```

Python Syntax

- Python also has built in support for **dictionaries**, allowing you to specify list indices with words or phrases (*keys*), instead of integers, which you were restricted to in C.
- But this creates a somewhat new problem... how do we iterate through a dictionary? We don't have indexes ranging from [0, *n*-1] anymore.

Python Syntax

- **Loops (redux)**

- The for loop in Python is extremely flexible!

```
for pie in pizzas:
    # use pie in here as a stand-in for "i"
```

Python Syntax

- **Loops (redux)**

```
for pie in pizzas:
   print(pie)
```

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}
```

```
cheese
vegetable
buffalo chicken
pepperoni
```

Python Syntax

- **Loops (redux)**

```
for pie, price in pizzas.items():
   print(price)
```

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}
```

Python Syntax

- **Loops (redux)**

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}
```

```
for pie, price in pizzas.items():
   print("A whole {} pizza costs ${}".format(pie, price))
```

```
A whole buffalo chicken pizza costs $12
A whole cheese pizza costs $9
A whole vegetable pizza costs $11
A whole pepperoni pizza costs $10
```

Python Syntax

- **Printing and variable interpolation**

- format gives one way to interpolate variables into our printed statements in a very printf-like way, but there are other/s.

```
print("A whole {} pizza costs ${}".format(pie, price))

print("A whole " + pie + " pizza costs $" + str(price))

# you may see this, but avoid; deprecated
print("A whole %s pizza costs $%2d" % (pie, price))
```

Python Syntax

- **Functions**

- Python has support for functions as well. Like variables, we don't need to specify the return type of the function (because it doesn't matter), nor the data types of any parameters (ditto).

- All functions are introduced with the def keyword.
  - Also, no need for main; the interpreter reads from top to bottom!
  - If you wish to define main nonetheless (and you might want to!), you must at the very end of your code have:

```
if __name__ == "__main__":
    main()
```

Python Syntax

- **Functions**

```
def square(x):
    return x * x
```

Python Syntax

- **Functions**

```
def square(x):
    return x ** 2
```

Python Syntax

- **Functions**

```
def square(x):
    result = 0
    for i in range(0, x):
        result += x
    return result

print(square(5))
```

Python Syntax

- **Objects**
- Python is an *object-oriented* programming language.
- An object is sort of analogous to a C structure.

Python Syntax

- **Objects**
- C structures contain a number of *fields*, which we might also call *properties*.
  - But the properties themselves can not ever stand on their own.

```
struct car
{
    int year;
    char *model;
}
```

```
struct car herbie;
herbie.year = 1963;
herbie.model = "Beetle";
```

Python Syntax

- **Objects**
- C structures contain a number of *fields*, which we might also call *properties*.
  - But the properties themselves can not ever stand on their own.
- Objects, meanwhile, have properties but also *methods*, or functions that are inherent to the object, and mean nothing outside of it. You define the methods inside the object also.
  - Thus, properties and methods don't ever stand on their own.

Python Syntax

- **Objects**
- C structures contain a number of *fields*, which we might also call *properties*.
  - But the properties themselves can not ever stand on their own.

```
struct car
{
    int year;
    char *model;
}
```

```
struct car herbie;
year = 1963;
model = "Beetle";
```

Python Syntax

- **Objects**

~~`function(object);`~~ `object.method()`

Python Syntax

- **Objects**

Python Syntax

- **Objects**
- You define a type of object using the `class` keyword in Python.
- Classes require an initialization function, also more-generally known as a *constructor*, which sets the starting values of the properties of the object.
- In defining each method of an object, `self` should be its first parameter, which stipulates on what object the method is called.

Python Syntax

- **Objects**

```
class Student():

    def __init__(self, name, id):
        self.name = name
        self.id = id

    def changeID(self, id):
        self.id = id

    def print(self):
        print("{} - {}".format(self.name, self.id))
```

```
jane = Student("Jane", 10)
jane.print()
jane.changeID(11)
jane.print()
```

Python Syntax

- **Style**
- If you haven't noticed, good style is **crucial** in Python.
- Tabs and indentation actually matter in this language, and things will not work the way you intend for them to if you disregard styling!
- Good news? No more curly braces to delineate blocks!
  - Now they just are used to declare dictionaries.

Python Syntax

- **Including files**
- Just like C programs can consist of multiple files to form a single program, so can Python programs tie files together.

~~`#include <cs50.h>`~~

Python Syntax

- **Including files**
- Just like C programs can consist of multiple files to form a single program, so can Python programs tie files together.

```
import cs50
```

Python Syntax

- **Including files**
- Just like C programs can consist of multiple files to form a single program, so can Python programs tie files together.

```
cs50.get_int()
cs50.get_float()
cs50.get_string()
```

Python Syntax

- Python programs can be prewritten in .py files, but you can also write and test short Python snippets using the Python interpreter from the command line.
- All that is required is that the Python interpreter is installed on the system you wish to run your Python programs on.

Python Syntax

- To run your Python program through the Python interpreter at the command-line, simply type

```
python <file>
```

- and your program will run through the interpreter, which will execute everything inside of the file, top to bottom.

Python Syntax

- You can also make your programs look a lot more like C programs when they execute by adding a **shebang** to the top of your Python files, which automatically finds and executes the interpreter for you.

```
#!/usr/bin/env python3
```

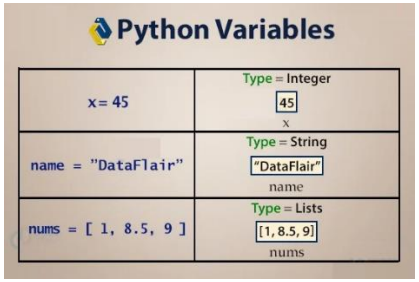
- If you do this, you need to change the **permissions** on your file as well using the Linux command `chmod` as follows:

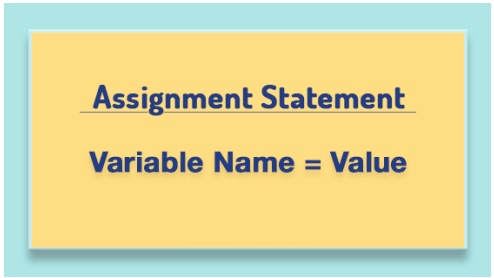
```
chmod a+x <file>
```

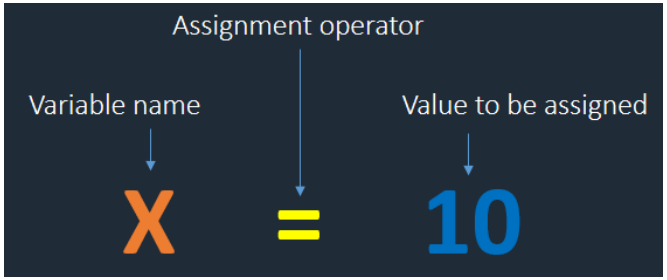
1. VARIABLES / DATA - what is the name, type and initVal?
2. BRANCHING - is it Boolean True or False?
3. LOOPS - control variable (sentry variable) -> while: what is the sentry initialization, condition, and increment/decrement? for: what is the sentry, start, finish and changes?
4. OPERATIONS
5. INPUT / OUTPUT - Input: where the answer is stored as variable? and what is the message? // Output: what is the message?
6. FUNCTIONS - what is the function_name, return type and parameters if any?
7. AGGREGATION

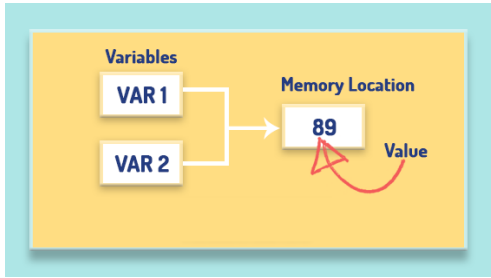
• The 7 big ideas: are main set of ideas which can be separated from specific languages, and it is simply about understanding those concepts, translating (converting) them into other languages (lines of code) & implement them •
• There are few main core ideas that tend to be universal to all traditional languages => all programming in any language comes down to below ideas (the 7 big ideas) •

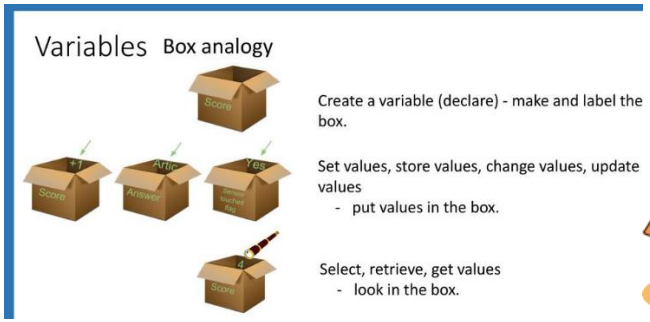
1. VARIABLES:

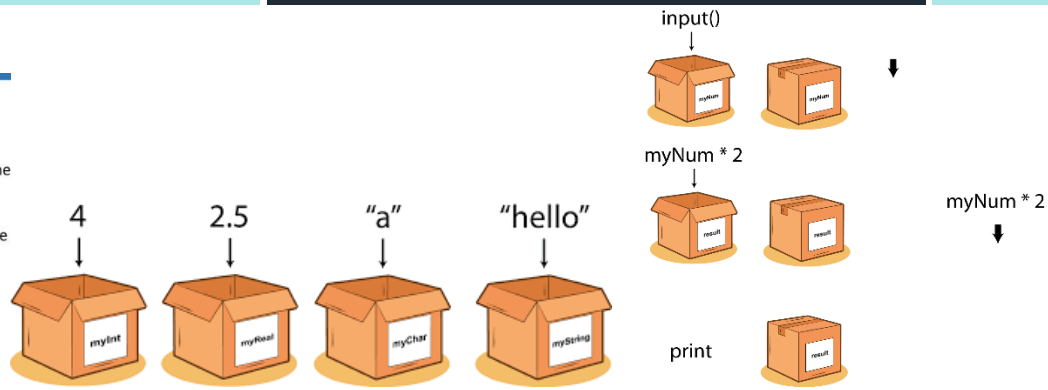












A. general info:

- programing is about data and data typically goes into variables => variables are a place in memory (home) to hold data => to find out the id of the memory (home) where the data is, use Python id() function that will return the "identity" of the object.
  -> the identity of an object is an integer, which is guaranteed to be unique and constant for this object during its lifetime: container = "hello" -> id(container)
  -> - first focus what computer sees = data then convert that to what the user sees

B. variables requirements are down to 3 questions:

I. what is the name of the var?
II. what is the type of the var?
III. what is the initial value (starting point) of that var?

B.1. a variable has a **name**, **usually one word** (make sure it is clear not just a letter), in python **variables names are like stickers put on objects.** Every **sticker** has a unique name written on it, and it can only be on one object at a time

-> rules to write a python variable (naming convention):

1. a variable name must start with a letter or underscore character

2. a variable name cannot start with a number

3. a variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _),using underscore between words is called **snake case (ex.this_is_snake_case, build_docker_image, call_python_method)**

4. variable names are case-sensitive (age, Age and AGE are 3 different variables)

5. variable can be written using **camelCase** (FileNotFoundException, toString)

I. algorithm must answer the question: what do we call this thing?

B.2. **variable is a container for a value and designed to hold a particular type of data and will behave also as that value, otherwise you will have to convert from one type to the other (in the fridge you have leftovers, it is good to put it in the right container)**

-> if type is different then you will have to use Type Cast (to cast a spell and to convert the data type of a value to another data type): y=int(y) #cast permanent spell// print(int(y)) #cast temporary spell// print("x is" + str(x)) <- so you need to have same values

-> you can assign one variable at a time or you can assign multiple data to multiple variables in the same time by using Multiple Assignment (put different food in multiple containers in the same time)

-> if you add more than one item into a var then that simple container will TRANSFORM into a container of tuple - with or without ( ), list - [ ], set - { } or dictionary - {:}

-> most common data types: integer, float, string, object (data is stored differently in memory)

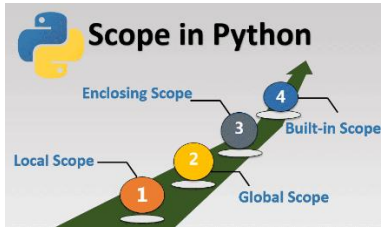
-> some languages are loose about types, some are restricted (python or JS does not matter if you do not mention), regardless, every variable has a type

II. algorithm must answer the question: what type of data does it contain?

B.3. a variable should always have a **starting initValue**

III. algorithm must answer the question: what is it's starting value?

C. how a variable can be accessed (point of access):



C.1. Scope: the region that a variable is recognized

C.1.1. globally: **available to be used globally and functions, class can access and ONLY read it too (you can have global and local var with same name)**

-> function CANNOT modify global variables unless you put inside the function the word 'global' in front of the var that was declared globally immediately after you create def

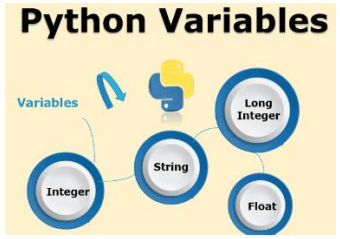
-> class CANNOT modify global variables unless you put inside the function the word 'global' in front of the var that was declared globally immediately after you create def

C.1.2. locally: **if declared inside the function then only the function can use it (you cannot access it from outside)**

C.2. class variable: **can be created within the class but outside the constructor**

-> you can set a default value for all instances of this class for all unique values created -> can be changed later if needed

D. variable data type methods:



- string method example:

a. string methods:

course.upper()

course.lower() <- turns capital beginning of the word and smaller the rest of the word

course.title()

b. string methods act as temporary converters:

course = 'Python for Beginners' -> print(course.upper()): will temporary modify the string -> when you print course it will be the original string not the upper case one

c. general purpose function:

len(course) <- count characters in a string

course.find()

course.replace()

'...' in course

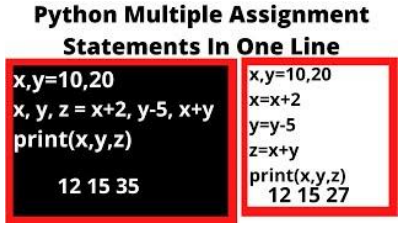
E. algorithm wording:

create a variable called _____ of type _____ with initial value _____

F. new variable algorithm example:

create a variable called name of type type that starts with the value iniVal: name = initVal

G. Multiple Assignment:




```
# Multiple assignment = allows us to assign multiple variables at the same time in one line of code

#Ex1:
#name="Bro"
#age=21
#attractive=True
name,age,attractive="Bro",21,True # same as above but grouped together, you have to respect the order
print(name)
print(age)
print(attractive)

#Ex2:
#Spongebob=30
#Patrick=30
#Sandy=30
#Squidward=30
Spongebob=Patrick=Sandy=Squidward=30
print(Spongebob)
print(Patrick)
print(Sandy)
print(Squidward)
```

H. Walrus Operator – PEP572:

- In the python 3.8 release, a new assignment operator was introduced which became wildly famous because of its being controversial and Guido van Rossum (creator of python) stepping down as BDFL (**Benevolent dictator for life**) “key visionary” “final decider” of the python language updates
- The Walrus operator :=
  - The name “walrus” because it resembles the eyes and tusk of a walrus

:=

- The update introduces a new syntax **:=** to assign value to a variable as a part of a larger expression
- Let's understand what does a "Walrus operator" do with an example:
  - Traditional way before python version 3.8:

```
my_list=[1,2,3,4,5,6]
print(my_list)
```

  - In the above code, a list is created first and then it is passed as an argument to the print function
    - Walrus way after python version 3.8

```
print(my_list:=[1,2,3,4,5,6]) #variable initialzed in the argument
print(my_list) #variable is saved
```

      - **By using the walrus operator you can avoid initializing a variable before passing it as an argument but instead, we can initialize it using the walrus operator right in the argument**
        - **Additionally, the variable initialized by the walrus operator can be reused further in the program**
- So, The walrus operator, introduced in Python 3.8, offers a way to accomplish two tasks at once: assigning a value to a variable, **and** returning that value, which can sometimes offer a way to write shorter, more readable code, that may even be more computationally efficient
  - simple assignment operator
    - We are all familiar with how to assign a value to a variable. We do so using the simple assignment operator:

```
num = 15
```

      And if we wanted to print the value of this variable using the ***print*** function, we can pass in the variable ***num*** as follows:

```
print(num)
# 15
```

      - Introduced in python 3.8, the walrus operator, (**:=**), formally known as the **assignment expression operator**, offers a way to assign to variables within an expression, including variables that do not exist yet
      - As seen above, with the simple assignment operator (**=**), we assigned **num = 15** in the context of a stand-alone statement
      - An expression evaluates to a value. A statement does something
      - In other words, the walrus operator allows us to both assign a value to a variable, **and** to return that value, all in the same expression. The name is due to its similarity to the eyes and tusks of a Walrus on its side ->>>> name := expr
        - **expr** is evaluated and then assigned to the variable **name**. That value will also be returned.
          - Let's look at some examples of the walrus operator in use:
            - **simple example:**
              - The best way to understand the walrus operator is with a simple example. Just like above, we want to assign 15 to **num**, and then print the value of **num**
                - We can accomplish both these tasks in one line of code using the walrus operator as follows:

```
print(num := 15)
# 15
```

                - The value of 15 is assigned to **num**. Then this same value is returned, which will become the argument for the **print** function. Thus, 15 is printed
                  - If we attempt the above using the simple assignment operator, we would get a **TypeError**, since **num = 15** does not return anything

```
print(num = 15)
# TypeError
```

            - **Walrus operator in while loops:**
              - To demonstrate how walrus operator can be used in while loop we will write a **program which stores all the input by the user until the user enters quit**
                - Traditional way

```
#list to store all inputs
list_of_inputs=[]

#ask input from user
x=input("Enter something")

#while loop checks if input is not quit
while (x!="quit"):
#append input in list of inputs list
 list_of_inputs.append(x)
#print list of inputs
 print(list_of_inputs)
#again asks the user to input something
 x=input("Enter something")
```

                - **Walrus way:**

```
#list to collect inputs
list_of_inputs=[]
```

```
#initialize a new variable x in every iteration until the user input quit
while (x:=input("Enter something")) != "quit":
#append the user input in list
  list_of_inputs.append(x)
#print the list of all inputs
  print(list_of_inputs)
```

- By using the walrus operator we have reduced the usage of writing the input function twice

- **Another example:**
  - Let's say that we want to keep asking a user for some input. If the user does not enter anything, then we want to stop asking for more inputs. We can do this using a while loop as follows:

```
value = input('Please enter something: ')

while value != '':
    print('Nice!')
    value = input('Please enter something: ')
```

```
Please enter something: Hello
Nice!
Please enter something: Great
Nice!
Please enter something: 25
Nice!
Please enter something: How are you?
Nice!
Please enter something:
```

  - *We ask the user to enter something, and assign that input to **value**. We then create a while loop that runs if the value entered is not an empty string*
    - *We print out 'Nice!' if the user successfully entered something. We then ask the user for another input and set that equal to **value**, and restart the process*
      - Let's now try this using the walrus operator:

```
while (value := input('Please enter something: ') != ''):
    print('Nice!')
```

```
Please enter something: Hello
Nice!
Please enter something: Great
Nice!
Please enter something: 25
Nice!
Please enter something: How are you?
Nice!
Please enter something:
```

      - *We ask the user for an input, and set that input equal to **value** using the walrus operator. This value is also returned, and compared to an empty string*
        - *If that comparison evaluates to **True** (not equal to an empty string), the code in the while loop runs, and 'Nice!' is printed. If it evaluates to **False**, the code does not run.*

- **Example with list comprehension:**
  - An example where the walrus operator can improve code readability and its computational efficiency is within list comprehensions that aim to filter out values
    - **List Comprehensions in Python:**
      - **A more elegant and concise way to create lists in python**
        - **For example**, let's say we have the list of numbers ***num_list***, and we want to add the cube of those numbers only if the cubed value is less than 20. We can do so as follows:

```
def cube(num):
    return num**3
```

```
num_list = [1,2,3,4,5]

[cube(x) for x in num_list if cube(x) < 20]
```

```
[1, 8]
```

        - *Notice how we have to call the function* **cube *twice***
          - The walrus operator will allow us to only call the function ***cube*** once in our list comprehension, as follows:

```
[y for x in num_list if (y := cube(x)) < 20]
```

```
[1, 8]
```

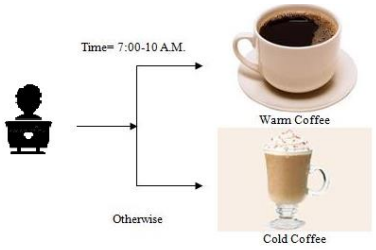
          - *The value of* **cube(x)** *is assigned to* **y**, *then returned and compared to 20. The value of* **y** *will only be added to the list if it is less than 20. Notice how the **cube()** function was only called once, leading to more efficient code*
            - *This efficiency improvement can become more substantial the more complex and computationally demanding the function is*

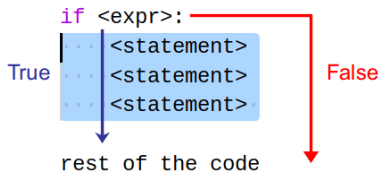
H. variable advanced topics:

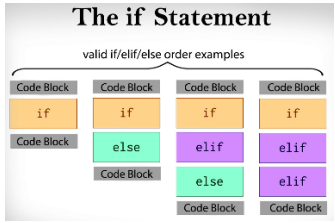
F.1. will have scope once functions and classes are introduced

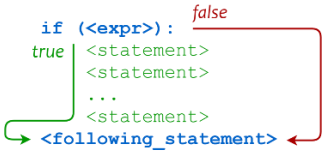
F.2. a class will become a new data type

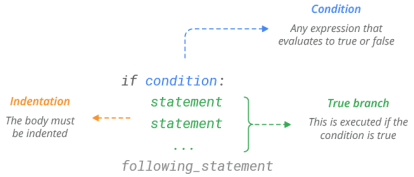
2. BRANCHING:

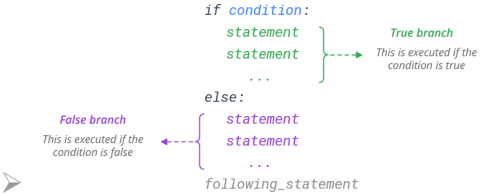


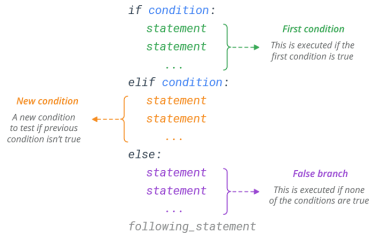


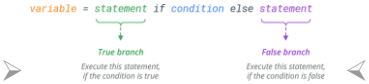












A. what is a condition:

- **a condition:** is an expression that can be evaluated as true or false expression, so, a condition is any expression that can be boiled down to **Boolean (true or false)** and that can be done through **comparison**
  - **-> if condition is true**, do one set of instructions
    - **-> if not true** you may have other instructions of if condition with **'else' clauses <- 'else' always to set up multiple conditions**
      - -> people make decisions every day, before you get outside you have an if statement that says if it's raining then I'd better take my jacket
      - -> computers are amazing once you decide those kinds of statements that they can reliably execute. so, computer program is a little bit of math and computer statements where the decisions gets made
      - **-> if statement is fundamental in computer programming = it is the key concept <- bill gates**
        - **-> conditions can also be nested inside each other -> will have complex set of behavior**
          - **-> there are variations, including switch / select <- python does not have switch statement**

B. illusions created by conditions:

- **condition would be a 2nd most important idea in programming after variables, a condition is some sort of true or false question**

- **it's the foundation because a computer seems to be smart**, appears to make decisions **BUT** they don't make decisions rather follow instructions based on a condition
  -> **condition is a behavior** that will happen **if the condition is met** and that's how computers appear to make those decisions, no matter how complicated it is, if you are talking about fancy AI or blockchain or neural networks it all comes down to this
    -> computers seem to be smart, but they are just storing the programmer's intelligence and playing it back

C. tools for branching (creating illusions of tree branching):

- **if - elif - else: using elif it becomes like a branch from a tree** (with elif you need to do a whole new condition), it is good to use else clause as in this way, you will know if you caught every possible value
  -> **if behavior has a shortcut evaluation:** so, one a condition is found true it does not look at the other conditions -> it shortcuts till the end
    -> **so, if you have 1 if -> 1 elif- > 1 else = it means you have 3 different values**

D. algorithm wording:

If condition is true, execute code

E. example of conditions:

if it's hot
  it's a hot day
  drink plenty of water
otherwise if it's cold
  it's a cold day
  wear warm clothes
otherwise
  it's a lovely day

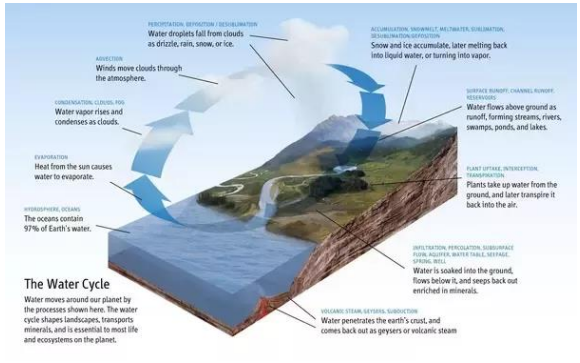
-->> we start by defining a Boolean variable is_hot = True <<--

```
is_hot = False
is_cold = True
if is_hot:
  print("it's a hot day")
  print("Drink plenty of water")
elif is_cold:
  print("it's a cold day")
  print("wear warm clothes")
else:
  print("it's a lovely day")
  print("enjoy your day")
```

F. branching advanced topics:

1. **exception-handling / event response**
2. **compound conditions (and / or)** -> **"else if"** acts just like an **"or" AND "nested if's"** act just like an **"and"** with a few other benefits

3. LOOPS:



A. general info:

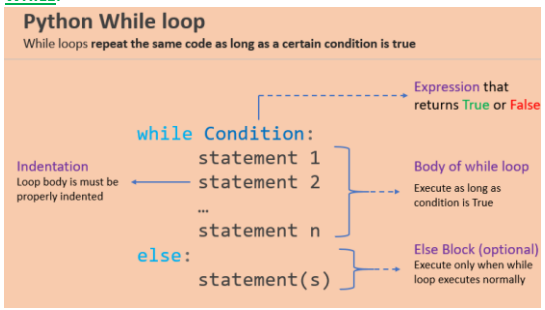
- **the two things computer are famous for is being able to appear to make decisions (that was branching) AND being able to repeat things and that's repetition**
- loops are also based on conditions but the important thing also is based on some sort of a **control variable** and typically we **called** it **sentry variable**, it **works like a guard decides who stays and who goes => sentry is the bodyguard of loops**

- loops are like copy/paste multiple times (instead of writing hello many times you will create a loop to do it for you in an instant) **<- Mark Zuckerberg**
- **computer manages repetition through loops: ->** all loops have some sort of **sentry (control) variable** that **will be the basis of that loop**
  -> the key to understanding any loop is to understand three things that all relate to this **sentry**, so every kind of loop (don't care what it is) is going to have **3 questions related to the behavior of the loop:**
  -> 1. how does loop start?
  -> 2. how does loop end?
  -> 3. how does sentry change to get to end condition?
  * **so, ultimately it is about the sentry variable, so a loop starts with sentry variable being at a particular value and it ends when the sentry variable is some different value and somehow it must change so to get from the beginning to the end**
    => that's the key to any kind of loop, so you must tie all above 3 questions correctly

B. loop types:

- **two main flavors of loop and they are keywords:**
  B.1. while (I DON'T know how many times something's going to happen)
  B.2. for (I KNOW how many times something's going to happen, either I know it directly or I'm going through a list that has a some sort of a set length)

B.1. WHILE:



B.1.1. while features:

B.1.1.1. non-deterministic: number of repetitions is unknown at start
B.1.1.2. has only 1 parameter: you must think about the sentry beforehand and **initialize the sentry before you got there**
-> so, plan ahead meaning you are planning algorithms, and somewhere inside the loop you have to change the sentry and at that point it will trigger the condition
B.1.1.3. end condition triggered by some state change:
* User input
* External event
* Anything that can be marked by a comparison
B.1.1.4. sentry can be any type
B.1.1.5. still check same three things (start, end, change)
B.1.1.6. while loops can also have else parts so at the end you can use else with while loop

B.1.2. while steps:

B.1.2.1. sentry: variable that will control loop
B.1.2.2. initialization code: code that initializes sentry

B.1.2.3. condition: loop repeats if condition is true
B.1.2.4. change code: code to change sentry so condition can be triggered

B.1.3. while algorithm wording:

- **Initialize sentry with initialization code then continue loop as long as condition is true. Inside loop, change sentry with change code**

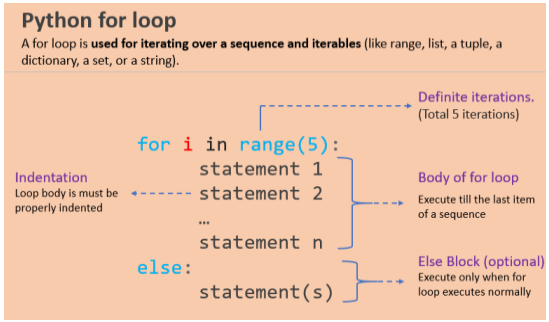
B.1.4. while loop general form:

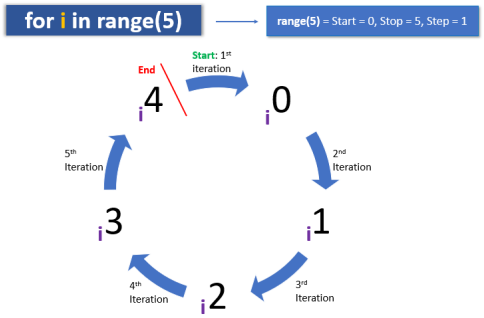
```
while condition:
  ... code will be repetitive
so:
i = 1:
while i <= 5:
  print(i)
  i = i + 1
print("Done")
```

B.1.5. while loop algorithm example:

```
""" keepGoing.py
demonstrate while loop with multiple exists
Andy Harris """

correct = "IndyPy"
tries = 0
keepGoing = True

while(keepGoing):

  tries = tries + 1
  print ("try # ", tries)
  guess = input("What is the password? ")

  if guess == correct:
    print("That's correct! here's the treasure!")
    keepGoing = False

  elif tries >= 3:
    print("Too many wrong tries, Launching the missiles")
    keepGoing = False
```

B.2. FOR:





B.2.1. for features:

B.2.1.1. deterministic: we know at the beginning how many times it will happen (ex. I will not talk out in class 100 times, so my counter will start at 0, it's going to 100, add 1 every time and will write 100 times same thing)
B.2.1.2. often something happens X times
B.2.1.3. also used to iterate some sort of list of things:

a. Items in a list
b. lines in a file
c. characters in a text variable

B.2.2. for steps:

B.2.2.1. sentry: integer variable that will control loop
B.2.2.2. start: integer value of sentry at beginning
B.2.2.3. finish: integer value of sentry at end
B.2.2.4. change: integer to add to sentry at each pass

B.2.3. for algorithm wording:
- begin with sentry at start and add change to sentry on each pass until sentry is larger than or equal to finish

B.2.4. for loop algorithm example:

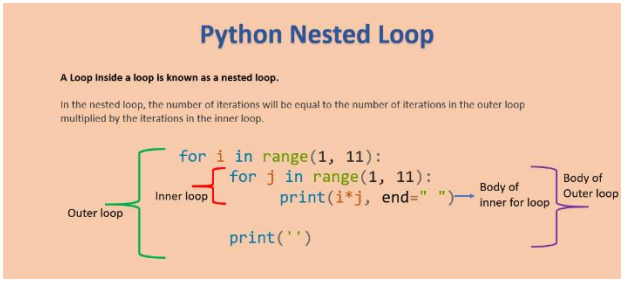
* Example 1 (strings):

```
""" will print bro code
with letters below each other """
for i in "Bro Code":
print(i)
```

* Example 2 (Integer):

```
""" countdown """
import time
for seconds in range (10,0,-1): // -1 to make it countdown
print(seconds)
time.sleep(1) / /will sleep for 1 second
print("Happy New Year!")
```

B.2.5. nested loop:



B.2.6. for advanced topics features: foreach, iterators

4. OPERATIONS:

A. general info:

- **operations:** are that what can you do with the data that you store in the variable (when you have data in memory you want to mess with it)

B. operations vary by data type:

1. **numeric:** add, subtract, multiply, divide, modulus, trig, etc
2. **text:** combine, search, split, iterate, transform

-> conversion: you can convert from one type to another using type casting

B.2.1. conversion algorithm wording:

- convert oldVariable to integer(string or float) and store in intVariable

C. advanced operations topics:

1. the methods of a class often become custom operations
2. you can redefine built-in operations

5. INPUT / OUTPUT (I/O):

A. general info:

- most programs don't just act on data, we need to get data from somewhere: the user or file, the internet and we need to put stuff somewhere -> the user file, internet database

I. INPUT:

I.1. input characteristics:

I.1.1. get information from the user or a file
I.1.2. implies an output
I.1.3. implies a variable to hold data (net to catch the ball)
I.1.4. may also involve a type conversion
I.1.5. in GUI style you will tend to have a label that should indicate what user should type in the box, but some output is implied

I.2. input requirements:

I.2.1. **message: question being asked of the user**
I.2.2. **variable: where answer from user will be stored**

-> when you have an input, you imply that there will be a question to the user, and you will use a net (variable to hold the answer) to catch the ball (user input) otherwise it will hit you in the head

I.3. input algorithm wording:

-> ask the user ____ (message) and store result in ____

I.4. input algorithm example:

Example:

""" ask the user message
and store the answer in variable """

variable = input ("message")

I.5. input advanced topics:

I.4.1. networking

I.4.2. databases that are forms of I/O

II. OUTPUT:

II.1. output characteristics:

II.1.1. print something to the user or a file

II.1.1. normally outputs text

II.2. output requirements:

II.2.1. often you need ways to combine multiple outputs and types

II.2.1. message: text to write to user

II.3. output algorithm wording:

-> output the text message ____

II.3. output algorithm example:

Example:
""" output the text message """
print ("message")

II.4. how to google an output topic:

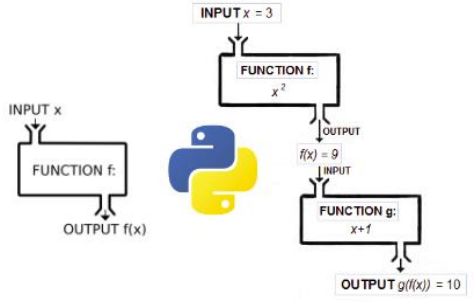
if you want to GOOGLE this: how do I output text in python3

↥ above can solve any problem (same was in basic long time ago, did not have other feature) ↥

modern language has other features to make less painful coding, next ideas will be about aggregation, about grouping things

↧ ↧

6. FUNCTIONS:



working with functions in python

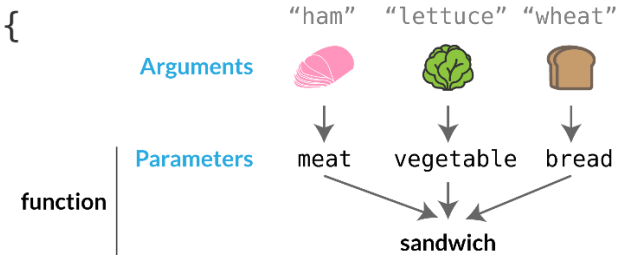
In Python, the **function is a block of code defined with a name**

- A Function is a block of code that only runs when it is called.
- You can pass data, known as parameters, into a function.
- Functions are used to perform specific actions, and they are also known as methods.
- **Why use Functions**? To reuse code: define the code once and use it many times.

```
def add(num1, num2):
    print("Number 1:", num1)
    print("Number 2:", num1)
    addition = num1 + num2

    return addition

res = add(2, 4)
print(res)
```

Function Name, Parameters, Function Body, Return Value, Function call

```
function makeSandwich( , , ){
    let sandwich =  +  +  ;

    return  ;
}
```




```
let meat1 = "ham";
let veg1 = "lettuce";
let bread1 = "wheat";

function makeSandwich(meat,vegetable,bread){

    let sandwich = meat + vegetable + bread;

    return sandwich;
}

let hamSandwich = makeSandwich(meat1, veg1, bread1);
// hamSandwich= "hamlettucewheat";
```

JavaScript Example

Python

```
meat_1 = "ham"
veg_1 = "lettuce"
bread_1 = "wheat"

def make_sandwich(meat, vegetable,bread):
    sandwich = meat + vegetable + bread
    return sandwich

ham_sandwich = make_sandwich(meat_1, veg_1, bread_1)
print(ham_sandwich)#hamlettucewheat
```

- **Let's think about the general concept of cooking with a recipe first. Using a recipe means that:**
  1. **You start with a specific set of ingredients**
  2. **You perform a specific procedure using those ingredients**
  3. **You will get a reliable product at the end**
     - A function is also a reusable recipe that performs the same set of actions over and over again on a set of ingredients
       - Those ingredients are called **parameters**
       - Some functions **return** a value. That means that they give you a new value that you can then use throughout your script. Other functions do not return a value
         - Instead, they might change a value that already exists in your script. Think of it like chopping onions. There is no "new" product, just the same product in a new format
       - Above we are **declaring** a make_sandwich function. It does return a value - a full sandwich. In other words, we are documenting the recipe on how to make a sandwich

A. function general info:

- python keywords: there is only a handful of keywords (commands) built into python => you take those primitive commands and you put them together to form new things

-> def is part of key commands and functions will have their own memory space

B. function definition:

- functions [Subroutines (a.k.a. routines or procedures)]**:** allow you to break a program up into more manageable little pieces **->** encapsulates tasks **->** If we do it right, these pieces work like building blocks that can be reused **->** enable easier shareability

-> we have already been using subroutines that are provided by Python: raw_input(), randint(), range(), etc. But as you know, we can also make our own

-> you solve a hard problem by smacking it with a sledgehammer until it becomes a bunch of little problems, and those smaller problems typically take our functions (actions)

-> each function will be a solution to a smaller problem with a group of related statements that performs a specific task, and then you solve all those little problems and put them back together make complex solutions

-> so, when we build a function we create new commands, like Lego set and we are building new Lego with our brain

C. function vs real life:

- in music we do labeling: chorus **...**, verse1**...**verse2**...**bridge**...,** the band will put together all this stuff, and then at the bottom of the sheet for a particular performance they will write **"road map"**

*** road map:** v1,v1,c then v1,v2,b,c then c,c,c so it says which part they want to do and in which order, you have words sheet where you write c,v1,v2....so **=>** we are labeling parts of the song c,v1,v2

-> 2 steps: first you define (def) the section of your song, then when you have them defined, you can put them in different order to have variation of the song

-> so, that means I'm about to define what it means to chorus (as example of a song), so you use the keyword def and then you will define the name of a function, and functions will always have **() + ":"**

-> means I'm gonna build a block of code, so what we are saying: **the indented code following me will be a function, what will**

Example 1. -> put the song in the function:

```
def verse1()
  print("how long does it have to be")
  print("you reckoning that you can't trust")
  print("how long does it have to be")
def verse2()
  print("if I just take, what you have said")
  print("it should have been so good together")
def verse3()
  print("don't you know, I know what you did")
  print("if you know, this crazy situation is above my imagination tonight")

def chorus ()
  print("ciao, ciao, senorita")
  print("Qui sera, sera tonight")

def verse4()
  print("tonight, everything is gonna be all right")
 def bridge()
  pass # instrumenta
```

-> def song road map:

```
verse1()
verse2()
verse3()
chorus()
verse4()
verse3()
chorus()
bridge()
chorus()
```

Example 2. Here's a very simple subroutine definition:

```
def sayGoodbye():
  print "Goodbye, my friend, goodbye. . . "
```

```
 return

Then here's where we use it (If we want to be fancy, we say we call or invoke it):
...
if userResp == "no":
 sayGoodbye(). . .
...
```

D. functions steps:

1. define a function def
2. put a comment after to say for what it is, so our function is like a miniature program, so in the same way we define or programs we define our function, we give a clue of what they do
3. call the function (and you can pass param through arguments from caller)

E. functions features:

1. allow you to group code to **'invent'** new commands **->** a function allows you to aggregate instructions
2. can take inputs (they do not have to do it, but they can)
3. can produce outputs (they do not have to do it, but they can)
4. allocate tasks to your function like you allocate tasks to people
5. **a function that doesn't return a value is like a new instruction (have the function DO a thing)**
6. **a function that does return a variable can generate new values (have the function MAKE a thing)**

F. functions variable scope:

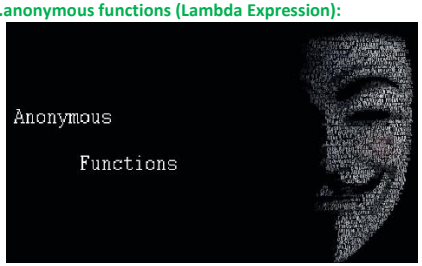
1. local variable function scope: variables created in a function, acts as local police (so inside a function is like its own little program **->** better to be used) **=** inside a function **=** finite lifetime
   -> functions are like Vegas (what happens in a function stays in a functions):as soon as my function is over my variable inside the function will go out of the memory and the computer won't remember it again
2. global variable function scope: variable created outside a function acts as state police (to be used for unusual cases) **=** introduce potential errors and you need to keep track of them

G. type of functions:

1. general purpose: len(),print(),input()
2. specific functions: example specific for strings, you access it with . , those are called also methods
3. standard functions:

```
def greet_user():
  print('Hi There')
  print('welcome aboard')
print("start")
greet_user() <- you cannot add the caller before the function
print("finish")
  -> output:
     Start
     Hi There
     welcome aboard
     finish
```

4.anonymous functions (Lambda Expression):



- **They are nameless functions**
- Not all functions have to have a name
- You can use lambda to sort, filter, or map etc over a list or tuple etc
- Sometimes we need to declare a function without any name. The nameless property function is called an **anonymous function** or **lambda function**
- **The reason behind using anonymous function is for instant use, that is, one-time usage**
  - **Normal function is declared using the** `def` **function. Whereas the anonymous function is declared using the** `lambda` **keyword**
    - **In opposite to a normal function, a Python lambda function is a single expression. But, in a lambda body, we can expand with expressions over multiple lines using parentheses or a multiline string**
      ex : `lambda n:n+n`

Syntax of `lambda` function:

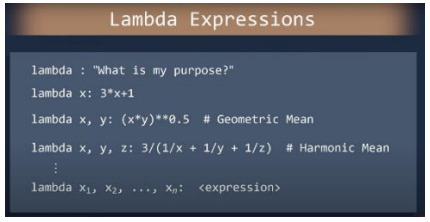
```
lambda: argument_list:expression
```

- **When we define a function using the lambda keyword, the code is very concise so that there is more readability in the code. A lambda function can have any number of arguments but return only one value after expression evaluation**

```
# without lambda
def f(x):
   return 3*x+1
print(f(2)) #7

# with lambda
#1. type keyword lambda - (lambda), lambda is not the name of the function, it is the python keyword that says what follows is an anonymous function
#2. inputs - (x)
#3. enter the expression that will be the return value - what is after (:)
print(lambda x : 3*x + 1) # this anonymous function will take the input x and return 3*x +1
                # <function <lambda> at 0x000001D8CA3B6A70> - this will print where it is stored as it has no name
g = (lambda x : 3*x + 1) # give it a name (g) so you can see result
print(g(2)) # 7
```

```
#lambda expression with multiple inputs
# imagine processing data from web registration form
# would like to combine first and last name to display on the user interface
# strip removes the leading and trailing white space
# title will make sure that only the first letter is capitalized

full_name = lambda fn, ln:fn.strip().title()+ " " + ln.strip().title()
print(full_name(" leonhard", "EULER")) # you use it just like any other function # Leonhard Euler
```

- **So, steps in writing a general lambda expression:**
  1. **Type the keyword lambda followed by 0 or more inputs (it is ok not have inputs)**
  2. **Type a colon (:)**
  3. **Type a single expression -> this expression is the return value, you cannot use lambda for multi-line expressions**



- **Example of when to use:**

```
# imagine we have a list of sci-fi authors, we would like to sort this list by last name
# some of theses authors have middle names while others have initials
# our mission is to create a lambda function that will extract the last name and use that as the sorting value
# list have a built-in method called sort, to see how to use it call it on the function name help(scifi_authors.sort)
scifi_authors = ["Issac Asimov", "Ray Bradbury", "Robert Heinlein", "Arthus C. Clarke", " Frannk Herbert", "Orson Scott Card", "Douglas Adams", "H. G. Wells", "Leigh Brackett"]

help(scifi_authors.sort)#sort(*, key=None, reverse=False)#the key argument is a function that will be used for sorting, we will pass it a lambda expression

# to access the last name, split the string into pieces where you have a last name, next access the last piece by index -1 and just for caution convert the string into lower case like this the sorting will not be case sensitive
scifi_authors.sort(key=lambda name: name.split(" ")[-1].lower())
print(scifi_authors)
```

- Let's see an example to print even numbers without a lambda function and with a lambda function. See the difference in line of code as well as readability of code:
  - **Example 1: Program for even numbers without lambda function:**

```
def even_numbers(nums):
   even_list = []
   for n in nums:
      if n % 2 == 0:
         even_list.append(n)
   return even_list
```

```
num_list = [10, 5, 12, 78, 6, 1, 7, 9]
ans = even_numbers(num_list)
print("Even numbers are:", ans)
```

Output

```
Even numbers are: [10, 12, 78, 6]
```

Example 2: Program for even number with lambda function:

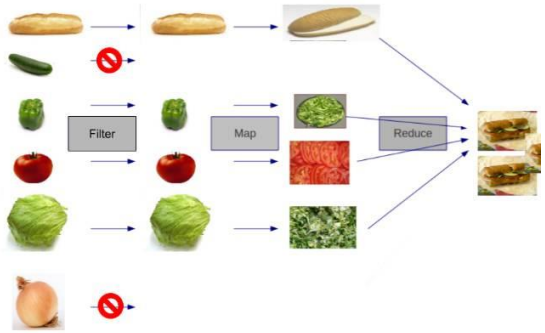
```
l = [10, 5, 12, 78, 6, 1, 7, 9]
even_nos = list(filter(lambda x: x % 2 == 0, l))
```

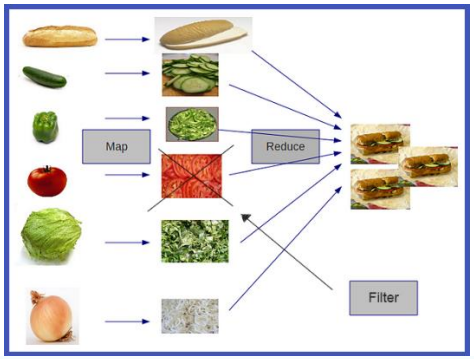
Output

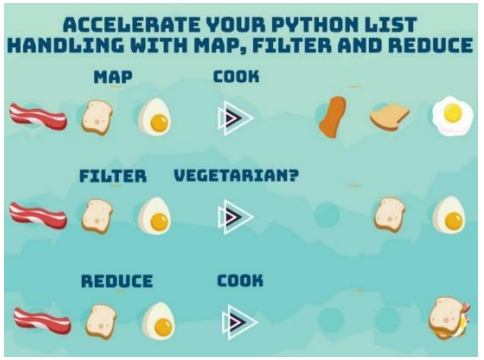
```
Even numbers are: [10, 12, 78, 6]
```

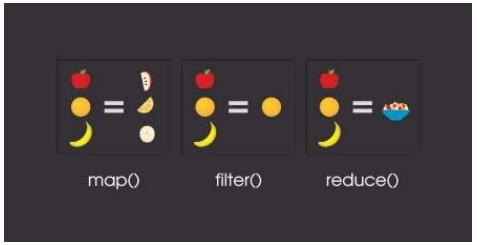
- We are not required to write **explicitly return statements** in the lambda function because the lambda internally returns expression value
- Lambda functions are more useful when we pass a function as an argument to another function. We can also use the lambda function with built-in functions such as filter, map, reduce because this function requires another function as an argument

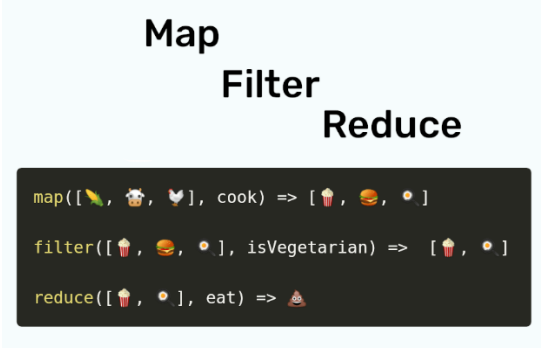
4.1. Map, Filter And Reduce In Pure Python:

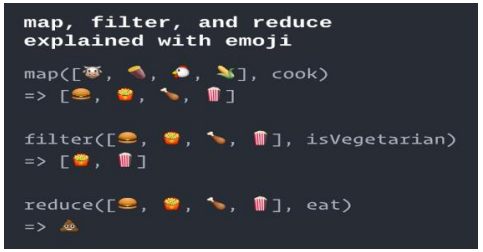


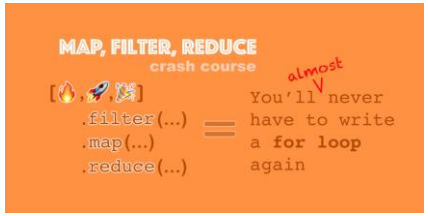


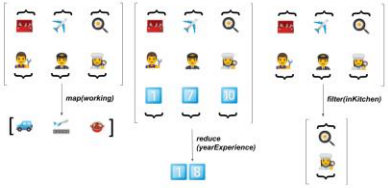


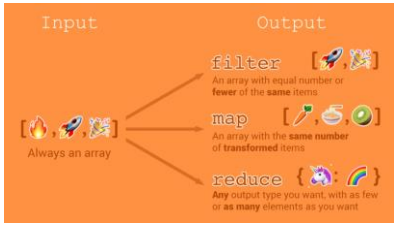












The Basics:

- **Map, filter and reduce are functions that help you handle all kinds of collections. They are at the heart of modern technologies such as Spark and various other data manipulation and storage frameworks**
  - But they can also very powerful helpers when working with vanilla Python

4.1.1. Map() function:

- In Python, the map() function is used to apply some functionality for every element present in the given sequence and generate a new series with a required modification

Ex: for every element present in the sequence, perform cube operation and generate a new cube list

Syntax of map() function:

```
map(function, sequence)
```

where,

- function – function argument responsible for applied on each element of the sequence
- sequence – Sequence argument can be anything like list, tuple, string

Example: lambda function with map() function

```
list1 = [2, 3, 4, 8, 9]
list2 = list(map(lambda x: x*x*x, list1))
print("Cube values are:", list2)
```

Output

```
Cube values are: [8, 27, 64, 512, 729]
```

- Map is a function that takes as an input a collection e.g. a list ['bacon','toast','egg'], and a function e.g. upper()
  - Then it will move every element of the collection through this function and produce a new collection with the same count of elements. Let's look at an example

```
map_obj = map(str.upper,['bacon','toast','egg'])
print(list(map_obj))
>>['BACON', 'TOAST', 'EGG']
```

  - What we did here is using the map(some_function, some_iterable) function combined with the upper function (this function capitalizes each character of a string)
  - As we can see we produced for every element in the input list another element in the output list
  - We receive always the same amount of elements in the output as we will put into it! Here we send 3 in and received 3 out, this is why we call it an N to N function. Let's look at how one can use it

```
def count_letters(x):
   return len(list(x))map_obj = map(count_letters,['bacon','toast','egg'])
print(list(map_obj))
>>[6, 5, 3]
```

  - In this example we defined our own function count_letters(). The collection was passed through the function and in the output, we have the number of letters of each string! Let's make this a little bit sexier using a lambda expression:

```
map_obj = map(lambda x:len(list(x)),['bacon','toast','egg'])
print(list(map_obj))
>>[6, 5, 3]
```

    - A lambda expression is basically just a shorthand notation for defining a function

4.1.2. Filter() function:

- In Python, the filter() function is used to return the filtered value. We use this function to filter values based on some conditions

Syntax of filter() function:

```
filter(function, sequence)
```

where,

- function – Function argument is responsible for performing condition checking.
- sequence – Sequence argument can be anything like list, tuple, string
  - **Example: lambda function with filter()**

```
l = [-10, 5, 12, -78, 6, -1, -7, 9]
positive_nos = list(filter(lambda x: x > 0, l))
```

```
print("Positive numbers are: ", positive_nos)
```

Output

```
Positive numbers are: [5, 12, 6, 9]
```

- In contrast to Map, which is an N to N function. Filter is a N to M function where N≥M. What this means is that it reduces the number of elements in the collection. In other words, it filters them!
  - As with map the notation goes filter(some_function, some_collection). Let's check this out with an example:

```
def has_the_letter_a_in_it(x):
  return 'a' in x# Let's first check out what happens with map
map_obj = map(has_the_letter_a_in_it,['bacon','toast','egg'])
print(list(map_obj))
>>[True,True,False]# What happens with filter?
map_obj = filter(has_the_letter_a_in_it,['bacon','toast','egg'])
print(list(map_obj))
>>['bacon', 'toast']
```

  - As we can see it reduces the number of elements in the list. It does so by calculating the return value for the function has_the_letter_a_in_it() and only returns the values for which the expression returns **True**
    - Again this looks much sexier using our all-time favorite lambda!

```
map_obj = filter(lambda x: 'a' in x, ['bacon','toast','egg'])
print(list(map_obj))
>>['bacon', 'toast']
```

4.1.3. Reduce() function:

- In Python, the reduce() function is used to **minimize sequence elements** into a **single value** by applying the specified condition
  - The reduce() function is present in the functools module; hence, we need to import it using the import statement before using it

Syntax of reduce() function:

```
reduce(function, sequence)
```

Example: lambda function with reduce()

```
from functools import reduce
list1 = [20, 13, 4, 8, 9]
add = reduce(lambda x, y: x+y, list1)
print("Addition of all list elements is : ", add)>/code>
```

Output

```
Addition of all list elements is : 54
```

- Let's meet the final enemy and probably the most complicated of the **3.** But no worries, it is actually quite simple. It is an N to 1 relation, meaning no matter how much data we pour into it we will get one result out of it
  - The way it does this is by applying a chain of the function we are going to pass it. Out of the 3, it is the only one we have to import from the functools
  - In contrast to the other two it can most often be found using three arguments reduce(some_function, some_collection, some_starting_value), the starting value is optional but it is usually a good idea to provide one. Let's have a look:

```
from functools import reducemap_obj = reduce(lambda x,y: x+" loves "+y, ['bacon','toast','egg'],"Everyone")
print(map_obj)
>>'Everyone loves bacon loves toast loves egg'
```

    - As we can see we had to use a lambda function which takes two arguments at a time, namely x,y. Then it chains them through the list. Let's visualize how it goes through the list
      1. x="Everyone", y=" bacon": return "Everyone loves bacon"
      2. x="Everyone loves bacon", y="toast": return "Everyone loves bacon loves toast"
      3. x="Everyone loves bacon loves toast", y="egg" : return "Everyone loves bacon loves toast loves eggs"
         - So we have our final element **"Everyone loves bacon loves toast loves eggs".** Those are the basic concepts to move with more ease through your processing pipeline
           - One honorable mention here is that you can not in every programming language assume that the reduce function will handle the element in order, e.g. in some languages it could be "'Everyone loves egg loves toast loves bacon'"

Combine:

- To make sure we understood the concepts let's use them together and build a more complex example:

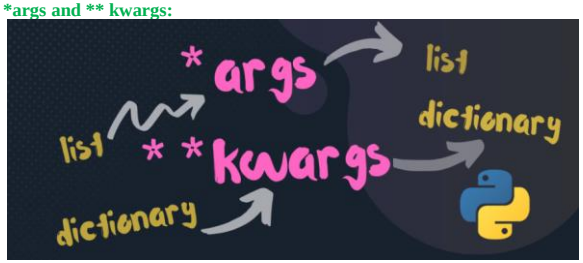
```
from functools import reducevals = [0,1,2,3,4,5,6,7,8,9]
# Let's add 1 to each element >> [1,2,3,4,5,6,7,8,9,10]
map_obj = map(lambda x: x+1,vals)
```

```
# Let's only take the uneven ones >> [1, 3, 5, 7, 9]
map_obj = filter(lambda x: x%2 == 1,map_obj)
# Let's reduce them by summing them up, ((((0+1)+3)+5)+7)+9=25
map_obj = reduce(lambda x,y: x+y,map_obj,0)
print(map_obj)
>> 25
```

- As we can see we can build pretty powerful things using the combination of the 3
  - Let's move to one final example to illustrate what this might be used for in practice. To do so we load up a small subset of a dataset and will print the cities which are capitals and have more than 10 million inhabitants!

```
from functools import reduce#Let's define some data
data=[['Tokyo', 35676000.0, 'primary'], ['New York', 19354922.0, 'nan'], ['Mexico City',
19028000.0, 'primary'], ['Mumbai', 18978000.0, 'admin'], ['São Paulo', 18845000.0, 'admin'],
['Delhi', 15926000.0, 'admin'], ['Shanghai', 14987000.0, 'admin'], ['Kolkata', 14787000.0,
'admin'], ['Los Angeles', 12815475.0, 'nan'], ['Dhaka', 12797394.0, 'primary'], ['Buenos Aires',
12795000.0, 'primary'], ['Karachi', 12130000.0, 'admin'], ['Cairo', 11893000.0, 'primary'], ['Rio
de Janeiro', 11748000.0, 'admin'], ['Ōsaka', 11294000.0, 'admin'], ['Beijing', 11106000.0,
'primary'], ['Manila', 11100000.0, 'primary'], ['Moscow', 10452000.0, 'primary'], ['Istanbul',
10061000.0, 'admin'], ['Paris', 9904000.0, 'primary']]map_obj = filter(lambda x:
x[2]=='primary' and x[1]>10000000,data)
map_obj = map(lambda x: x[0], map_obj)
map_obj = reduce(lambda x,y: x+", "+y, map_obj, 'Cities:')
print(map_obj)
>> Cities:, Tokyo, Mexico City, Dhaka, Buenos Aires, Cairo, Beijing, Manila, Moscow
```

5. *args and ** kwargs:



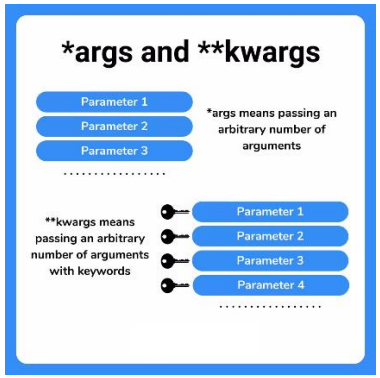
Arbitrary Arguments are often shortened to *args and **Arbitrary Keyword Arguments** are often shortened to **kwargs in Python documentations

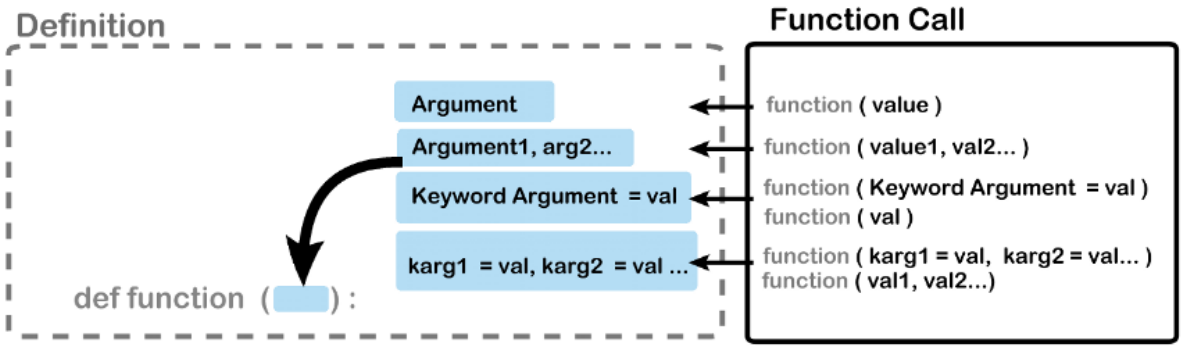
What do they do?

- In short **args & **kwargs allow you to pass an **undetermined amount (* & **)** of arguments and named arguments (**args** = arguments & **kwargs** = key-worded arguments)
  - Before fully explaining them, we need to take a step back and review some basics. Let's start with functions (*we'll check classes later*)

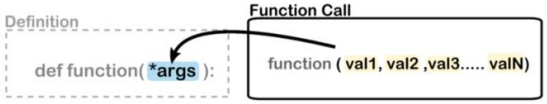
```
# A simple function with one argument or positional argument...def simpleFunction(arg):
  return( 'I am a function, this is my argument: ' + arg )print(simpleFunction('Argy'))>>>I am a function, this is my argument: Argy
      ---------- • ----------
# Now let's try key-worded arguments and 3 ways to add them when calling a function:def simpleFunctionNamedArgs(arg1='Argy', arg2=2):
  return( 'I am a function, these are my named arguments: arg1=' + arg1 + ', arg2=' + str(arg2))  ---------- 1 ----------print(simpleFunctionNamedArgs())>>> Default Arguments:
I am a function, these are my named arguments: arg1=Argy, arg2=2  ---------- 2 ----------
print(simpleFunctionNamedArgs('Argidio', 4))>>> Sequential Arguments
I am a function, these are my named arguments: arg1=Argidio, arg2=4
      ---------- 3 ----------print(simpleFunctionNamedArgs(arg2=6, arg1='Argos'))>>> Named Arguments, note the order doesn't matter as long as the keywords fit the function definition:I am a function, these are my named arguments: arg1=Argos, arg2=6
```

▸ Summarized :





5.1. Arbitrary Arguments (*args) - Undetermined number of arguments *args:



▸ The magic/usefulness of *args is that you can quickly make a function that takes 1, or 1,000 arguments and do something with them. Here are some **examples:**

```
# Simple function that receives multiple arguments and returns them:
def multiArguments(*args):
   return(args) #returns a tuple
print (multiArguments('one', 'two', 'three') ) # args returns a tuple, which is ordered and unchangeable...
>>>
('one', 'two', 'three')
   ---------- • ----------

# In order to do something useful with the arguments, we need to "unpack" them, here's the classical way:
def multiArguments(*args):
   for arg in args:
      print(arg + 2)
multiArguments(10, 20, 30)
>>>
12
22
32
   ---------- • ----------

# And here with comprehension
def multiArgumentsComprehension(*args):
    print([arg+100 for arg in args])
multiArguments(100, 200, 300)
>>>
[200, 300, 400] # we get a list in this last example
```

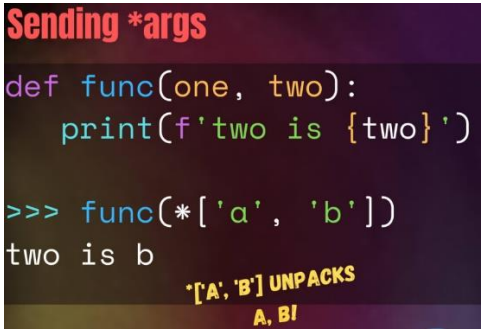
• If we are not sure with number of arguments that will be passed in our function, we can use * before the parameter name so that the function will receive a **tuple** of arguments, and we can access the items accordingly

```
#function
def welcome( *match ):
   print("\n Total number of matches are : ",len(match))
   for n in match:
```

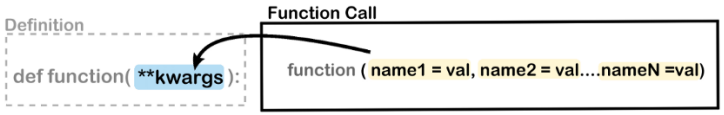
```
    print("Score :",n )


# Passing 9 parameters
welcome(10,9,8,9,7,6,9,8,5)

# Passing 6 parameters
welcome(4,5,9,4,10,5)
```

Output :

```
 Total number of matches are :  9
Score : 10
Score : 9
Score : 8
Score : 9
Score : 7
Score : 6
Score : 9
Score : 8
Score : 5

 Total number of matches are :  6
Score : 4
Score : 5
Score : 9
Score : 4
Score : 10
Score : 5
```



5.2. Arbitrary Keyword Arguments (**kwargs) - Undetermined number of key-worded arguments **kwargs:



- Conceptually this is the same as with *args, except here we are dealing with key-worded arguments. Hence **kwargs but requires a slightly different way to unpack:

```
# Simple function that receives multiple key-worded arguments and returns them, notice it returns a key, value dictionary:
def multiKeywordedArguments(**kwargs):
   return(kwargs)
print(multiKeywordedArguments(kwarg1 = 'uno', kwarg2 = 'dos', kwarg3 = 'tres'))
>>>
```

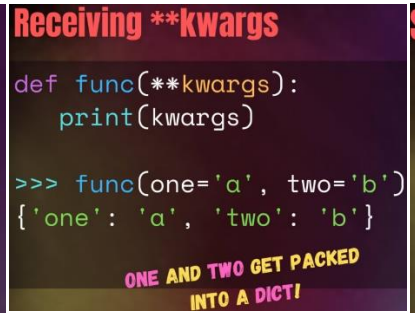
```
{'kwarg1': 'uno', 'kwarg2': 'dos', 'kwarg3': 'tres'}
        ---------- • ----------

# Unpacking kwargs much like args can be done with a for loop:
def multiKeywordedArguments(**kwargs):
  for name,value in kwargs.items():
    print('name: ' + name + ' value: ' + value)
multiKeywordedArguments(kwarg1 = 'uno', kwarg2 = 'dos', kwarg3 = 'tres')
>>>
name: kwarg1 value: uno
name: kwarg2 value: dos
name: kwarg3 value: tres
        ---------- • ----------

# and you can also use dict comprehension (note we need to make a copy):
def multiKeywordedArgumentsComprehension(**kwargs):
  modifiedDict = {key:value + 100 for (key,value) in kwargs.items()}
  print(modifiedDict)
multiKeywordedArgumentsComprehension(kwarg1 = 100, kwarg2 = 200, kwarg3 = 300)
>>>
{'kwarg1': 200, 'kwarg2': 300, 'kwarg3': 400}
```

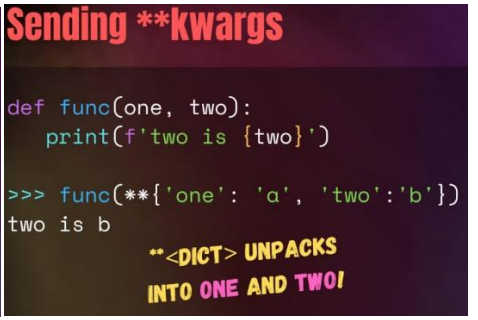
• If we are not sure with number of keyword arguments that will be passed in our function, we can use ** before the parameter name so that the function will receive a **dictionary** of arguments, and we can access the items accordingly

```
#function
def details( **name ):
   print("First Name : " + name["fname"])

# Passing 2 parameters
details(fname="Muskan", lname="Agarwal")

# Passing 1 parameter
details(fname="Anchal")
```

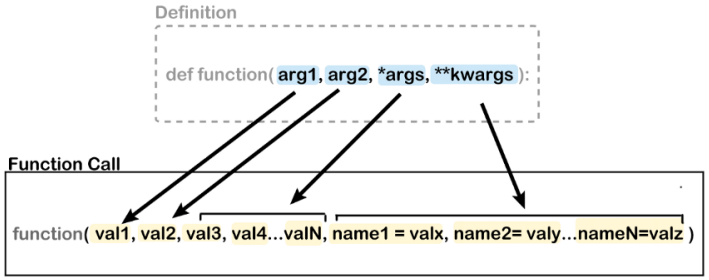
Output :

```
First Name : Muskan
First Name : Anchal
```





5.2. Mix and Match:



- Most likely you will encounter both *args and **kwargs in the same function definition. Usually in function templates, this is a Python programming convention. Here are a couple of common examples:

```
def mixAndMatch(*args, **kwargs):
   print(f' Args: {args}' )
   print(f' Kwargs: {kwargs}' )
mixAndMatch('one', 'two', arg3 = 'three', arg4 = 'four')
>>>
Args: ('one', 'two')
Kwargs: {'arg3': 'three', 'arg4': 'four'}
Note that you cannot place them in the wrong order on the function definition, for instance  (**kwargs, *args) will give you an error
        ---------- • ----------

# And if you want to add regular arguments they go before *args or **kwargs
def mixAndMatch(regArg, *args, **kwargs):
   print(f' Regular Argument: {regArg}' )
   print(f' Args: {args}' )
   print(f' Kwargs: {kwargs}' )
mixAndMatch('zero', 'one', 'two', arg3 = 'three', arg4 = 'four')
>>>
Regular Argument: Zero
Args: ('one', 'two')
Kwargs: {'arg3': 'three', 'arg4': 'four'}
```

- **Classes:**
  - Another place where *args and **kwargs can be found is in classes and subclasses:

```
# Here we are passing an undetermined number of arguments and key-worded arguments to a class/object upon creation, and also to a method/function inside the class.class simpleclass(object):
   def __init__(self, *args, **kwargs):
      print (args, kwargs)
   def classMethod(self, *args, **kwargs):
      print (args, kwargs)
classInstance = simpleclass('one', 'two', arg3 = 'three', arg4 = 'four')
>>>
('one', 'two') {'arg3': 'three', 'arg4': 'four'}
classInstance.classMethod('uno', 'dos', arg3 = 'tres', arg4 = 'cuatro')
>>>
('uno', 'dos') {'arg3': 'tres', 'arg4': 'cuatro'}
        ---------- • ----------

# Another place where they usually appear is in subclassing, let's say we want to pass arguments to the parents init method: class parentClass(object):
   def __init__(*args, **kwargs):
      print('parentArgs:', args, kwargs)

class childClass(parentClass):
   def __init__(self, *args, **kwargs):
      print('childArgs:', args, kwargs)
      parentClass.__init__(*args, **kwargs)
subclassInstance = childClass('one', 'two', arg3 = 'three', arg4 = 'four')
childArgs: ('one', 'two') {'arg3': 'three', 'arg4': 'four'}
parentArgs: ('one', 'two') {'arg3': 'three', 'arg4': 'four'}
```

- **Extra credit: Unpacking with * & ****
  - A related topic to *args and **kwargs is unpacking lists, tuples, and dictionaries with * and **, which can be considered the inverse of what we have been doing. Consider the following examples:

```
# Trying to pass a list or tuple as an argument fails without the unpacking operators...def needsUnpacking(arg1, arg2,arg3):
  print('My arguments:', arg1, arg2, arg3)
commonTuple = ('One','Two', 'Three')
commonList = ['uno','dos', 'tres'] # needsUnpacking(commonList) >>> ERROR
# needsUnpacking(commonTuple) >>> ERRORneedsUnpacking(*commonList)
>>>
My arguments: uno dos tresneedsUnpacking(*commonTuple)
My arguments: One Two Three
        ---------- • ----------
# Same use for Dictionaries...def needsUnpacking(arg1='ONE', arg2='TWO',arg3='THREE'):
  print('My arguments:', arg1, arg2, arg3)
commonDict = {'arg1':'un', 'arg2':'deux', 'arg3':'trois'}
needsUnpacking(commonDict)
>>>
My arguments: {'arg1': 'un', 'arg2': 'deux', 'arg3': 'trois'} TWO THREE # Doesn't throw an error but is INCORRECTneedsUnpacking(**commonDict)
>>>
My arguments: un deux trois # This is what we wanted
```

- You should be able to figure out how to read most * and ** in Python from now on. What remains is figuring out when to use them in your own code...

The quick answer is that it depends on what you are coding. There are genuine cases where you don't know how many arguments your function or class will be receiving (*you can also abuse this syntax easily*). Your code might be more readable if you are more explicit with your arguments.A reasonable compromise is to add *args and **kwargs to your functions while prototyping and once you figure out what arguments you need, then add them explicitly to your function definitions

H. passing information to a function:

```
def feed_me(food):  <-- this will act like a local variable that we define inside the function like below, you provide information to a subroutine by passing values to it
  print ("me like", food)
  return
feed_me("krill")
feed_me("hamburgers")
```

- **"krill" & "hamburgers "are the arguments and "food" is the parameter**
- **parameters = food, holes, or place holders for the subroutine (function) that we define for receiving information (think of the parentheses as the subroutine's mouth)**
- **arguments = the actual pieces of information that we supply (feed) to these functions to fill those holes**
- **when you define subroutine parameters, you're saying "Here is the information that this subroutine needs to do its job", and you're giving each piece of information a name so that the subroutine can use it**
- **when you define a subroutine, you must provide a list of any parameters that it requires**
- **we say that the values for the parameters are passed to the subroutine. The parameters are identified by their position (or order) inside the parentheses when the subroutine is called**
- **within a subroutine, the parameters act like variables that get their initial values from whatever was passed in.**
- **but like a bubble popping, those variables disappear when the subroutine returns.**
- **for example, when this call is made: find_volume(ln, 35, 2*num) then inside the subroutine, length will have the value of the variable ln, width will have the value 35, and height will have a value 2 times the value of the variable num**
- **More on Parameters:**
  - **subroutines can have multiple parameters:**

```
def find_volume(length, width, height):
  vol = length * width * height
  print "the volume is", vol
  return
```

I. type of subroutines:

- a subroutine has a job –> that is, it does something for us. There are **two kinds of subroutines:**

1) some subroutines just do their job, and don't give us back a value

```
def greet_user():
  print('welcome aboard')

greet_user()
```

2) some subroutines do something, and give us back a value that we can use later on in our program

- **we say that they return a value**
- **subroutines that return values are frequently called functions**

2.1. Example 1. numsBy3List = range(1,100,3)
Example 2. userResp = raw_input("Play again?")
Example 3. numToGuess = random.randint(1,20)
Example 4. print "There are", len(myList), "items"

2.2. return analogy (Kirby Urner metaphor!):

- **if subroutines can eat, it stands to reason that they can also, shall we say, poop*.**
- **we've already seen this kind of thing with raw_input(),which poops a string that the user typed, and range(), which poops a list of integers**
- that is, they can give you something back after they've done their job, more specifically, they can return a value

```
def fahrenheitToC(fahrenheitTemp ):
  retval = (fahrenheitTemp –32) * 5 / 9.0
  return retval <- Means "return this value back to whoever called us"
```

➢ temp = fahrenheitToC(212) **<- whatever fahrenheitToC() returned will become the new value of temp**

☆ **return examples:**

Example 1. below subroutine makes an <h1> heading for

```
def makeHtmlHeading(text): <- the parameter text
  return "<h1>" + text + "</h1>"
```

Example 2. below subroutine returns the larger of the two parameters

```
def max(val1, val2):
  if val1 > val2:
    return val1
  else:
    return val2
```

2.3. subroutines can return Boolean values –True or False:

```
def startsWithVowel(word):
  if word[0] in "aeiouAEIOU":
    return True
  else:
    return False
```

2.3.1. If the subroutine returns True or False, it can be used in a condition for an if or while statement:

```
userWord = raw_input("Gimme a word:")
if startsWithVowel(userWord):
  print userWord + "starts with a vowel")
else:
  print userWord + "does not start with a vowel"
```

2.4. why return when you can print?

- There is a big difference between a subroutine printing a result and returning a result
- when a subroutine prints a result, it is displayed to the human user, but that does not give the code that called the subroutine access to the value – the subroutine is a ***"black box"*** to the rest of the program
- when a subroutine returns a result, that value is available to whatever called it – to be saved in a variable, used for further calculation, and so forth

J. function algorithm wording:

- create a function called functionName ____ passing the values parameters _____ that returns a value of return type

K. functions example:

- **ex.** what did you do on a summer vacation?
  - **->** list of 3,4 things (had overdue parties because we couldn't meet before, I went on a hiking trip, I spent time with my son
  - **->** I went on a hiking trip **=** that involves a lot of pieces. there's a lot of details to that **SO** a function will allow you to take an entire set of instructions and put them in a bag and give them a name and ultimately allows you to create new commands in your language
  - **->** when you save the name, you are creating a magic spell, it will follow those instructions

L. functions advanced topics:

1. recursion
2. first-place status
3. lambdas (anonymous functions)

7. AGGREGATE DATA:

A. definition:

why choosing the right Data Structure is ***'extremely'*** important**?**

- data Structures are specific ways of organizing data and storing data; so that they can be accessed and worked within an efficient way
- data Structures define the relationship between the data and the operations which can be performed on it
- choice of a data structure depends on the problem we are solving and the kind of data we have
- inappropriate data structure can lead to unnecessarily long running times, memory leaks and waste of storage — which could translate to loss of a million dollars, a few days of time; when we talk about data on scale
- **if you have more than 1 data into the variable, then that variable will TRANSFORM and will wrap everything together in a shape of tuple, list, set, dictionary -> that var to aggregate data**

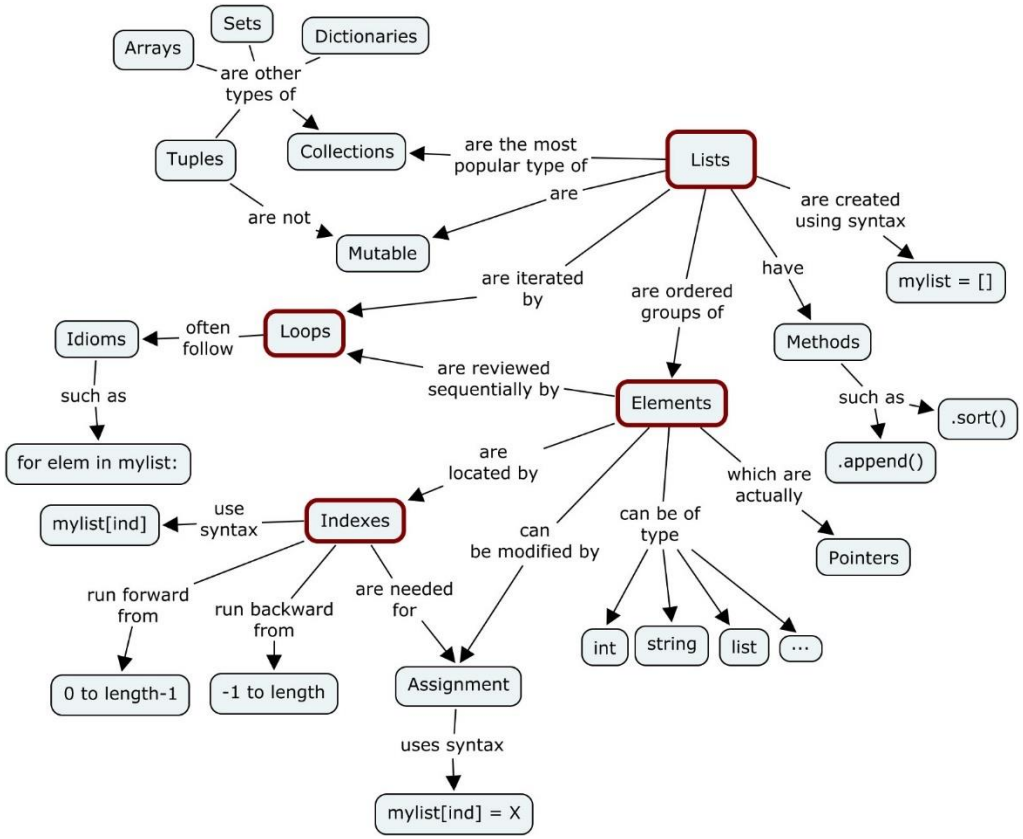
B. general info:

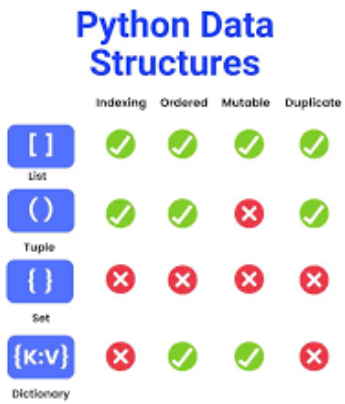
- programming these days is about data, having a lot of data means we can do something with it **->** key to modern software engineering
  - **why Tuples are Preferred over Lists:**
    - **Tuples are preferred when the user does not want the data to be modified. Sometimes, the user can create an object that is intended to remain intact during its lifetime**
    - **Tuples are immutable, so they can be used to prevent accidental addition, modification, or removal of data**
    - **Also, tuples use less memory, and they make program execution faster than using lists. Lists are slower than tuples because every time a new execution is done with lists, new objects are created, and the objects are not interpreted just once**
    - **Tuples are identified by Python as one immutable object. Hence, they are built as one single entity**

C. data structures comes in many forms:

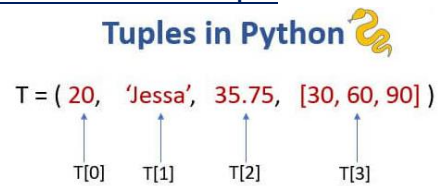
- **Data structures: The basic Python data structures in Python includes list, set, tuples, and dictionary**
- **Data structures are "containers" that organize and group data according to type**
- **One of the differing points among the data structures is mutability, which is the ability to change an object after its creation**
- **Lists and tuples are the most useful data types, and they can be found in virtually every Python program**
- Each of the data structures are unique in their own way

C.1. tuple
C.2. lists & 2d lists
C.3. set
C.4. dictionaries
C.5. arrays
C.6. classes & OOP





C.1. Python Data Structures – Tuples:



- ✓ **Ordered**: Maintain the order of the data insertion.
- ✓ **Unchangeable**: Tuples are immutable and we can't modify items.
- ✓ **Heterogeneous**: Tuples can contains data of types
- ✓ **Contains duplicate**: Allows duplicates data

- **Lists are mutable, whereas tuples are immutable**
- **2 dimensional lists are extremely powerful as they have lots of application in data science and machine learning**
- **Tuples cannot be modified, added, or deleted once they've been created ->** Once created, cannot be updated
- A tuple is a built-in data structure in Python that is an **ordered collection of objects** surrounded by parenthesis, rather than bracket
- **Unlike lists, tuples come with limited functionality**
- **The primary differing characteristic between lists and tuples is mutability ->** heterogeneous immutable sequence
- Lists are defined by using parentheses to enclose the elements, which are separated by commas
- The use of parentheses in creating tuples is optional, but they are recommended to distinguish between the start and end of the tuple
- A **sample tuple** is written as follows: **tuple_A = (item 1, item 2, item 3,..., item n)**

- Empty and One Single Item Tuple: When writing a tuple with only a single element, the coder must use a comma after the item
  -> This is done to enable Python to differentiate between the tuple and the parentheses surrounding the object in the equation
  -> A tuple with a single item can be expressed as follows: some_tuple = (item 1,)
  -> **If the tuple is empty, the user should include an empty pair of parentheses as follows: Empty_tuple= ( )**

  - **General form ⤵**

```
>>> a = (5, 6.54, "United States")
>>> type(a)
<class 'tuple'>
```

  - **Can be accessed using 0 based indices ⤵**

```
>>> a[1]
6.54
```

  - **Return the length using len() ⤵**

```
>>> len(a)
3
```

  - **Can contain any object, and thus can be nested ⤵**

```
>>> b = (1, 1.25, ("US", "UK"))
>>> b[2][0]
'US'
```

  - **For a single element tuple, trailing comma has to be used ⤵**

```
>>> b = (a, )
>>> type(b)
<class 'tuple'>
```

  - **Parentheses can be omitted in most of the cases ⤵**

```
>>> b = 1, 1.25, 'US'
>>> type(b)
<class 'tuple'>
```

  - **A tuple can be unpacked to variables ⤵**

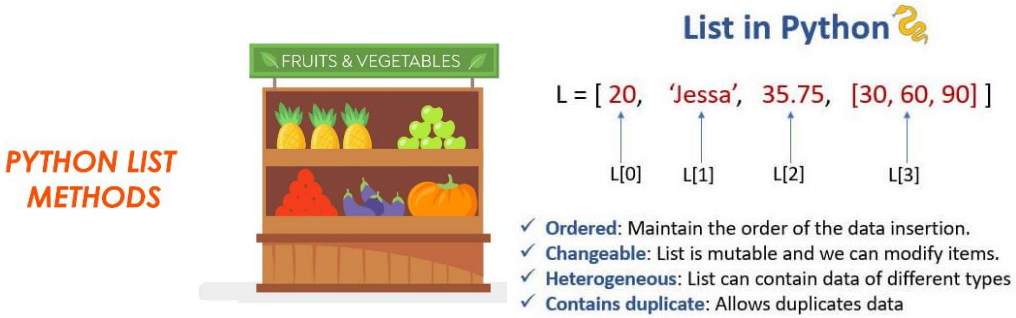
```
>>> a, b = (5, 6)
>>> a
5
>>> b
6
# Swapping 2 elements using unpacking
>>> a = 5
>>> b = 10
>>> a, b = b, a
```

  - **A tuple constructor can be used to create tuple from other iterables ⤵**

```
# Creating a tuple from a list
```

```
>>> a = [1, 2, 3, 4, 5]
>>> type(a)
<class 'list'
>>>> b = tuple(a)
>>> type(b)
<class 'tuple'>
>>> b
(1, 2, 3, 4, 5)
# Creating a tuple from a string
>>> tuple('strings')
('s', 't', 'r', 'i', 'n', 'g', 's')
```

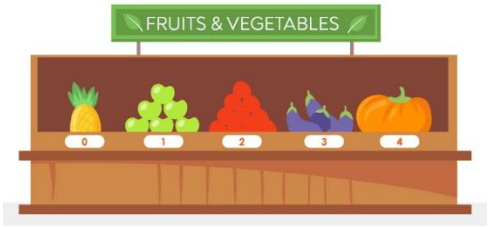
C.2. Python Data Structures – Lists:



- A list is defined as an mutable **ordered collection of items**, and it is one of the **essential data structures when using Python** to create a project
- **The term "ordered collections" means that each item in a list comes with an order that uniquely identifies them**
- The order of elements is an inherent characteristic that remains constant throughout the life of the list
- Since **everything in Python is considered an object**, creating a list is essentially creating a Python object of a specific type
- When creating a list, all the items in the list should be put in square brackets and separated by commas to let Python know that a list has been created

```
#LIST
fruits = ['pineapple', 'apples', 'tomato', 'eggplant', 'pumpkin']
fruits
```

```
['pineapple', 'apples', 'tomato', 'eggplant', 'pumpkin']
```



- **Lists can be nested**: A list can be nested, which means that it can contain any type of object
  - **->** It can include another list or a sublist – which can subsequently contain other sublists itself
    - **->** There is no limit to the depth with which lists can be nested. An example of a nested list is as follows:
      List_A = [item 1, list_B, item 3….., item n]
- **Lists are mutable**, Lists created in Python qualify to be mutable because **they can be altered even after being created**
- **A user can search, add, shift, move, and delete elements from a list at their own will**
- When replacing elements in a list, the number of elements added does not need to be equal to the number of elements, and Python will adjust itself as needed
- It also allows you to replace a single element in a list with multiple elements
- **Mutability also enables the user to input additional elements into the list without making any replacements**
- Enclosed on squared brackets:

```
>>> l = ['A', 'Quick', 'Brown', 'Fox']
```

Operations:

- **Add item to a list ↴**

```
# ADD ITEM USING NAME OF THE ITEM
fruits = ['pineapple', 'apples', 'tomato', 'eggplant', 'pumpkin']
fruits.append('watermelon')
fruits
```

```
['pineapple', 'apples', 'tomato', 'eggplant', 'pumpkin', 'watermelon']
```

- **Index from the end ↴**

```
>>> l = ['A', 'Quick', 'Brown', 'Fox']
>>> l[1]
'Quick'
# last element>>> l[-1]
'Fox'
# second last element
>>> l[-2]
'Brown'
```

```
fruits = ['pineapple', 'apples', 'tomato', 'eggplant', 'pumpkin']
fruits.pop(2)
fruits
```

```
['pineapple', 'apples', 'eggplant', 'pumpkin']
```

- **Slicing a list ↴**

```
>>> l = ['A', 'Quick', 'Brown', 'Fox']
>>> slice = l[:3]
>>> slice
['A', 'Quick', 'Brown']
>>> slice = l[1:3]
>>> slice
['Quick', 'Brown']
```

- **Content vs Object equality ↴**

```
>>> l = ['A', 'Quick', 'Brown', 'Fox']
# creates a new list with contents of l
```

```
>>> full_slice = l[:]
>>> full_slice
['A', 'Quick', 'Brown', 'Fox']
# Checking if contents of 'l' and 'full_slice' are same
>>> l == full_slice
True
# Checking if 'l' and 'full_slice' refer to same object in memory
>>> l is full_slice
False
```

- **Repeating List using * operator ↴**

```
>>> a = [1, 0]
>>> a * 4
[1, 0, 1, 0, 1, 0, 1, 0]
```

- **Find the index of an element ↴**

```
>>> l = ['A', 'Quick', 'Brown', 'Fox']
>>> l.index('Brown')
2
# if item is not present in the list
>>> l.index('jumps')Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: 'jumps' is not in list
```

- **Remove element from a list ↴**

```
# remove by index using del keyword
>>> l = ['A', 'Quick', 'Brown', 'Fox']
>>> del l[1]
>>> l
['A', 'Brown', 'Fox']# remove by item
>>> l = ['A', 'Quick', 'Brown', 'Fox']
>>> l.remove('Quick')
>>> l
['A', 'Brown', 'Fox']
```

```
# REMOVE ITEM USING NAME OF THE ITEM
fruits = ['pineapple', 'apples', 'tomato', 'eggplant', 'pumpkin']
fruits.remove('tomato')
fruits
```

```
['pineapple', 'apples', 'eggplant', 'pumpkin']
```

- **Insert item in the list ↴**

```
>>> l = ['A', 'Quick', 'Brown', 'Fox']
# insert at an index
>>> l.insert(1, 'Healthy')
>>> l
['A', 'Healthy', 'Quick', 'Brown', 'Fox']
```

```
# ADD ITEM USING INDEX
fruits = ['pineapple', 'apples', 'tomato', 'eggplant', 'pumpkin']
fruits.insert(3,'watermelon')
fruits
```

```
['pineapple', 'apples', 'tomato', 'watermelon', 'eggplant', 'pumpkin']
```

- **Extending a list ↴**

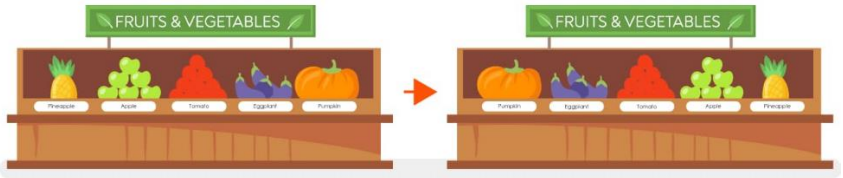
```
>>> a = [1, 2, 3]
>>> a = a + [4, 5]
>>> a
[1, 2, 3, 4, 5]
# Inplace using +=
>>> a = [1, 2, 3]
>>> a += [4, 5]
>>> a
[1, 2, 3, 4, 5]
# Inplace using extend
>>> a.extend([4, 5])
>>> a
[1, 2, 3, 4, 5]
```

- **In-place reverse and sort on a list ↴**

```
# inplace reverse
>>> a = [1, 2, 3]
>>> a.reverse()
>>> a
[3, 2, 1]
# inplace sort
>>> a.sort()
>>> a
[1, 2, 3]
```

```
# REVERSE LIST
fruits = ['pineapple', 'apples', 'tomato', 'eggplant', 'pumpkin']
fruits.reverse()
fruits
```

```
['pumpkin', 'eggplant', 'tomato', 'apples', 'pineapple']
```



- **In-place sorting a list with an explicit key function ↴**

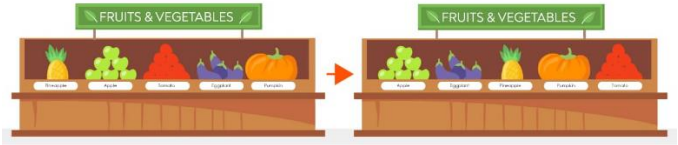
```
# sorting with a key - accepts a function for producing a sort key
>>> l = 'A quick brown box jumps over the lazy dog'.split()
>>> l
['A', 'quick', 'brown', 'box', 'jumps', 'over', 'the', 'lazy', 'dog']
# sorts according the length of the words
>>> l.sort(key=len)
>>> l
['A', 'box', 'the', 'dog', 'over', 'lazy', 'quick', 'brown', 'jumps']
```

- **Using sorted() to return a sorted list, keeping the original unmodified ↴**

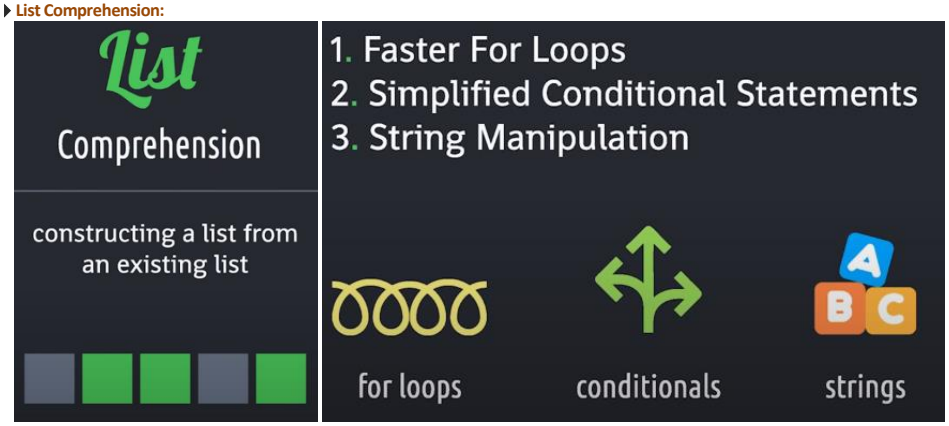
```
>>> a = [1, 4,2, 7]
>>> b = sorted(a)
>>> a
[1, 4, 2, 7]>>> b
[1, 2, 4, 7]
```

```
# SORT LIST
fruits = ['pineapple', 'apples', 'tomato', 'eggplant', 'pumpkin']
fruits.sort()
fruits
```

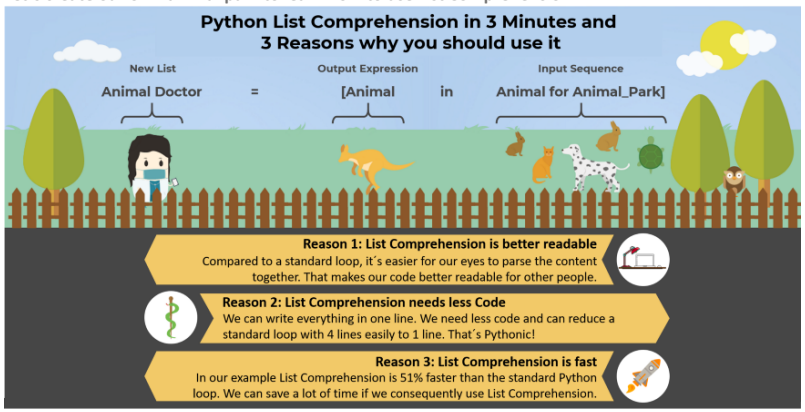
```
['apples', 'eggplant', 'pineapple', 'pumpkin', 'tomato']
```



List Comprehension:



- Let´s create our own animal park to learn how to use List Comprehension



- List comprehension is a powerful method to create new lists from existing lists. If you start using Python, List Comprehension might look complicated but you should get familiar with it as fast as you can
  - You can select specific elements from a list to create a new one. You can compare different lists and pick the elements you need. You can even use Pandas Series or Numpy Arrays with List Comprehension. Let´s have a deeper look!

What is List Comprehension?

- If you have an existing list you might pick some elements to create a new list. You could do this with a standard Python loop but with List Comprehension you will definitely need less code
- Lists can be iterated over which makes these objects so handy. Iteration is just an expression for the fact that you can go through the list one by one
  - Imagine you have a animal park with different animals. Once in a year each animal needs to get a check from the animal doctor. The animal park is our list
    - We can go through the animal park and pick animals one by one. The animal doctor is our new list who receives our iterables which are in this case animals

```
# Creating our animal park
animal_park = ['Rabbit','Rabbit','Rabbit','Rabbit',
'Cat','Cat','Cat','Cat','Cat','Cat','Cat',
'Turtle','Turtle','Turtle','Turtle','Turtle','Turtle','Turtle',
'Dog','Dog',
'Kangaroo','Kangaroo','Kangaroo','Kangaroo','Kangaroo','Kangaroo']

#Creating a new list for our animal doctor with all animals
animal_doctor = []
for animal in animal_park:
  animal_doctor.append(animal)
```

- Our code is very simple. First we created our animal park and after we created our animal doctor. In a next step we go through our animal park and pass each animal 1 by 1 to our animal doctor
  - Now we rewrite this loop as List Comprehension:

```
animal_doctor = [animal for animal in animal_park]
```

- You can see that we reduced the amount of lines from 3 to 1. If we compare both versions, we can see that we do exactly the same:

Comparison of the standard loop and List Comprehension

- **Conditional statements — Today we just want to check the predators:**
  - There are a lot of cases in which we want to use conditional statements. In our standard loop it could look like this:

```
animal_doctor = []
for animal in animal_park:
  if animal != 'Dog' and animal != 'Cat':
    animal_doctor.append(animal)
```

  - As you can see, we have to add another line of code for the conditional statement. Our animal doctor is just checking the predators today. We can do the same with our list comprehension:

```
animal_doctor = [animal for animal in animal_park if animal != 'Dog' and animal != 'Cat']
```

  - We can still write the whole expression in one line and it´s better readable than the loop version
- **If you are not convinced yet — List Comprehension is fast:**
  - We saw that list comprehension needs less code and is better readable. Let´s compare the speed of the standard loop with our list comprehension:

```
%%timeit
vet = [animal for animal in animal_park if 'o' in animal]
```

```
2.86 µs ± 14.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%%timeit
vet = []
for animal in animal_park:
    if 'o' in animal:
        vet.append(animal)
```

```
4.32 µs ± 8.41 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Comparison of List Comprehension and a standard loop in terms of speed

  - As you can see, List Comprehension is 51% faster than the standard loop. If you can use List Comprehension to create a new list from another list, always use it. There is no reason to use the slower standard loop

- **String manipulation:**


```
my_string = "HelloMyNameIsMariya"

my_string = "".join(
   [i for i in my_string]
   ) #HelloMyNameIsMariya
print(my_string)

my_string = "".join(
   [i if i.islower() else " " + "babana" if i in ["N","I"] else " " + i for i in my_string]
   )[1:]# to remove space at index 0
print(my_string)
```

- **Conclusion:**
  - We saw that List Comprehension is easy to use if the basic concept is clear

- It is a mighty tool to create new lists from other lists and should always be used if you want to create a new list from one or more existing lists: It´s better readable than the standard loop, it needs less code and it´s faster
- In pythn lists are collection of data surrounded by brackets and elements are separated through comas but instead of a list of data inside you enter an expression followed by a for loops and if classes
  List:[1,2,"a",3.14]
  [expr for val in collection]
  - The first expression generates the elements in the list and you follow this with a for loop for some collection of data this will evaluate the expression for every item in the collection
    - If you only want to include the expression for certain pieces of data you can an if class after the for loop, the expression will be added to the list only if the if clause is true and you can have more than if's
      [expr for val in collection if<test>]
      [expr for val in collection if <test1> and <test2>]
- You can loop even more than 1 collection
  [expr for val1 in collection1 and val2 in collection2]

Example1:

```
#list squares of the first 100 integers

# without list comprehensions

squares = []
for i in range(1, 101):
   squares.append(i**2)
print(squares)

# using list comprehensions
#1.we will call list comprehensions: squares2
#2.open a bracket and type the expression for each term in the list: i**2 (what you want it to do)
#3.enter the desired for loops :for i in range(1,101)
squares2 = [i**2 for i in range(1,101)]
print(squares2) # same result as above with 1 line of code instead of 3
```

Example2:

```
>>> remainders5 = [x**2 % 5 for x in range(1, 101)]
>>>
```

$$x ** 2 \ \% \ 5 \longleftrightarrow x^2 \pmod 5$$

```
#create a list of remainders when you divide the first 100 squares by 5
# to find the reminder when you divide by 5 use the % operator
reminders5 = [x**2 % 5 for x in range(1,101)]
print(reminders5)

reminders11 = [x**2 % 11 for x in range(1,101)]
print(reminders11)
```

Example3:

```
# list comprehensins with if clause
# if we have a list of movies and we want to find those movies that starts with the letter G

movies = ["Star Wars", "Ghandi", "Casablanca", "Shawshan Redemption"," Toy Story", "Gone with the Wind", "Citizen Kane", "It's a Wonderfull Life","The wizzard of Oz","Gattaca","Rear Window","Ghostbusters","Good Will
Hunting","2001: A Space Odyssy","Raiders of the Lost Ark","Groundhog Day","Close Encounters of the Third Kind"]

#withut list comprehensions:
gmovies = []

for title in movies:
   if title.startswith("G"):
      gmovies.append(title)
print(gmovies)

# the bove 4 lines can be done in 1 line
# so loop over the movies but also check that the title starts with the letter g
```

```
gmovies = [title for title in movies if title.startswith("G")]
print(gmovies)# same result as above ['Ghandi', 'Gone with the Wind', 'Gattaca', 'Ghostbusters', 'Good Will Hunting', 'Groundhog Day']
```

Example4:

```
#we have a list of movies is a list of tuples containing both the tile of the movie and the year it was released
# we want a list of the titles of the movies that we release before the year 2000

movies =[("Citizen Kane",1941),("Spirited Away",2001),("It's a Wonderful Life",1046),("Gattaca",1997),("No Country for Old Men",2007),("Rear Window",1954),("The Lord of the Rings:The Fellowship of the Ring",2001),("Groundhog Day",1993),("Close Encounters of the Third Kind",1997),("The Royal Tenebaums",2001),("The Aviator",2004),("Raiders of the Lost Ark",1981)]

pre2k = [title for (title,year) in movies if year < 2000] # the year was included in the list but just the title was included in the list
print(pre2k)
```

Example5:

```
names = ["mariya", "BATMAN", "spongebob"]
new_names = []

for n in names:
    if n.islower():
        n = n.capitalize()
    else:
        n = "Relax " + n.capitalize()
    new_names.append(n)

names = new_names
print(names)

>> ['Mariya', 'Relax Batman', 'Spongebob']
```

```
names = ["mariya", "BATMAN", "spongebob"]

[
n.capitalize() if n.islower()
else "Relax " + n.capitalize()
for n in names
]

>> ['Mariya', 'Relax Batman', 'Spongebob']
```

```
my_string = "hi442nm233ag2"

new_string = "".join(
    [ "d" if i=="4"
      else "e" if i=="2"
      else "s" if i=="3"
      else i
      for i in my_string ]
    )

print(new_string)

>> hiddenmessage
```

Example6:

- Print(i) = the body of the for loop or the actions which we would like to perform on all the list item (you can add the for loop first then add the action what you want to apply to that list)

```
fruits = ["apples", "bananas", "strawberries"]
```

for loop

```
for i in fruits:
    print(i)
```

list comprehension

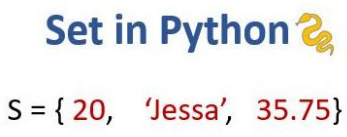
```
[ print(i) for i in fruits ]
```

```
fruits = ["apples","bananas","strawberries"]

# without list comprehension
new_fruits = []

for fruit in fruits:
   fruit = fruit.upper()
   new_fruits.append(fruit)

print(new_fruits)

# with list comprehension
fruits = [fruit.upper() for fruit in fruits]
print(fruits)# same resut as above ['APPLES', 'BANANAS', 'STRAWBERRIES']
```

Example with if:

```
bits = [False, True, False, False, True, False, False, True]

# without list comprehension
new_bits = []
for b in bits:
   if b == True:
      new_bits.append(1)
   else:
      new_bits.append(0)
print(bits) # [False, True, False, False, True, False, False, True]
print(new_bits) # [0, 1, 0, 0, 1, 0, 0, 1]

# with list comprehensions
super_bits = [1 if b == True else 0 for b in bits]
print(super_bits)# same as above [0, 1, 0, 0, 1, 0, 0, 1]
```

C.3. Python Data Structures – Sets:



- ✓ **Unordered**: Set doesn't maintain the order of the data insertion.
- ✓ **Unchangeable**: Set are immutable and we can't modify items.
- ✓ **Heterogeneous**: Set can contains data of all types
- ✓ **Unique**: Set doesn't allows duplicates items

- An unordered collection of unique, immutable objects - **that do not follow a specific order**
- enclosed by **{}**
- use set() to create an empty set
- a common use is to efficiently remove duplicates
- **Sets are used when the existence of an object in a collection of objects is more important than the number of times it appears or the order of the objects**
- **Unlike tuples, sets are mutable – they can be modified, added, replaced, or removed**
- **We can apply operations like intersection, union, issubset, issuperset on a set object**
- A **sample set** can be represented as follows: **set_a = {"item 1", "item 2", "item 3",….., "item n"}**
- One of the ways that sets are used is when checking whether or not some elements are contained in a set or not
  -> For example, sets are highly optimized for membership tests. They can be used to check whether a set is a subset of another set and to identify the relationship between two sets

- **Create a set ↴**

```
>>> s = {1, 2, 3, 5}
>>> s
{1, 2, 3, 5}
```

- **Create a set from other collection ↴**

```
>>> l = [1, 2, 2, 3, 4, 4, 5]
# create a set from a list
>>> s = set(l)
>>> s
{1, 2, 3, 4, 5}
```

- **Add an element to a set ↴**

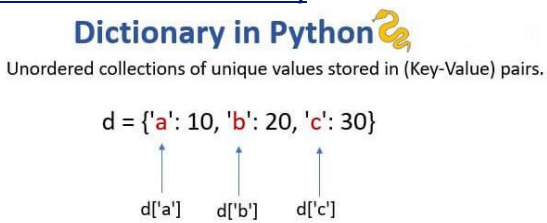
```
>>> s = {1, 2, 3, 4}
```

```
>>> s
{1, 2, 3, 4}
>>> s.add(6)
>>> s
{1, 2, 3, 4, 6}
# adding an already existing element doesn't have any effect, also it doesn't give # error
>>> s.add(6)
>>> s
{1, 2, 3, 4, 6}
# adding multiple elements in one go with update
>>> s.update([7, 8, 9, 10])
>>> s
{1, 2, 3, 4, 6, 7, 8, 9, 10}
```

- **Delete an element from a set ↴**

```
# delete with remove() - gives error if element is not present
>>> s
{1, 2, 3, 4, 6, 7, 8, 9, 10}
>>> s.remove(1)
>>> s
{2, 3, 4, 6, 7, 8, 9, 10}
>>> s.remove(1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 1
# with discard() - no error even if element is not present
>>> s
{2, 3, 4, 6, 7, 8, 9, 10}
>>> s.discard(1)
```

C.4. Python Data Structures – Dictionary:



- ✓ **Unordered**: The items in dict are stored without any index value
- ✓ **Unique**: Keys in dictionaries should be Unique
- ✓ **Mutable**: We can add/Modify/Remove key-value after the creation

- A dictionary = Unordered mapping of unique immutable keys to mutable values
- Keys must be unique in a dictionary
- use mustache { } for dictionary
- it's about key, value pairs(key is associated with pairs):

Name: John Smith
Email: john@gmail.com
Phone: 1234

```
customer = {
   "name":"john smith",
   "age":30,
   "is_verified":True
    } # <- print(customer.get("name")) # <- use get to get the value ("name" is the key) and value otherwise the prompt will say None
```

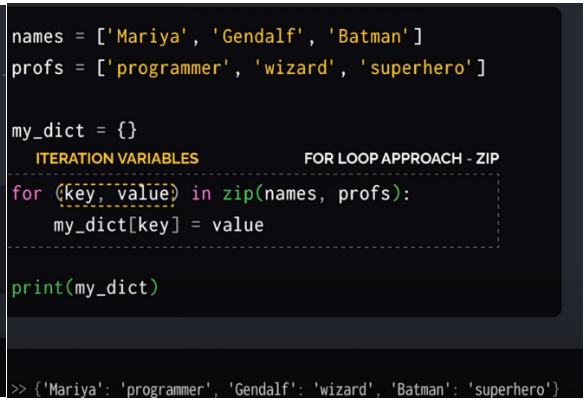
- **comma-separated key-value pairs joined by a colon ↴**

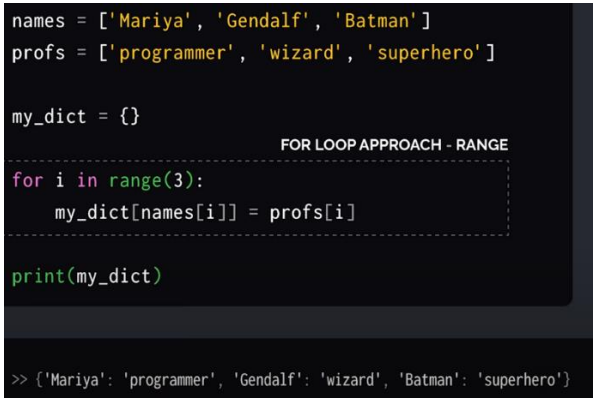
```
>>> d = {'Google': 'www.google.com', 'Facebook': 'www.facebook.com'}
>>> d
{'Google': 'www.google.com', 'Facebook': 'www.facebook.com'}
# access with a key
>>> d['Google']
'www.google.com'
```

- **Iterating a dictionary ↴**

```
>>> for k, v in d.items():
...     print(f'{k} -> {v}')
...
Google -> www.google.com
Facebook -> www.facebook.com
```

- **dictionary comprehension ↴**





Zip APPROACH

```
names = ['Mariya', 'Gendalf', 'Batman']
profs = ['programmer', 'wizard', 'superhero']

my_dict = {
key:value for (key, value) in zip(names, profs)
}
```

```
>> {'Mariya': 'programmer', 'Gendalf': 'wizard', 'Batman': 'superhero'}
```

Range APPROACH

```
names = ['Mariya', 'Gendalf', 'Batman']
profs = ['programmer', 'wizard', 'superhero']

my_dict = {
names[i]:profs[i] for i in range(3)
}
```

```
>> {'Mariya': 'programmer', 'Gendalf': 'wizard', 'Batman': 'superhero'}
```

```
names = ['Mariya', 'Gendalf', 'Batman']
profs = ['programmer', 'wizard', 'superhero']
```

zip approach

```
{key:value for (key, value) in zip(names, profs)}
```

range approach

```
{names[i]:profs[i] for i in range(len(names))}
```

```
{'Mariya': 'programmer', 'Gendalf': 'wizard', 'Batman': 'superhero'}
```

```
names = ('Mariya', 'Gendalf', 'Batman')
profs = ('programmer', 'wizard', 'superhero')
```

zip approach

```
{key:value for (key, value) in zip(names, profs)}
```

range approach

```
{names[i]:profs[i] for i in range(len(names))}
```

```
{'Mariya': 'programmer', 'Gendalf': 'wizard', 'Batman': 'superhero'}
```

```
names = pd.DataFrame(['Mariya', 'Gendalf', 'Batman'])
profs = pd.DataFrame(['programmer', 'wizard', 'superhero'])
```

zip approach

```
{key:value for (key, value) in zip(names[0], profs[0])}
```

range approach

```
{names[0][i]:profs[0][i] for i in range(len(names))}
```

```
>> {'Mariya': 'programmer', 'Gendalf': 'wizard', 'Batman': 'superhero'}
```

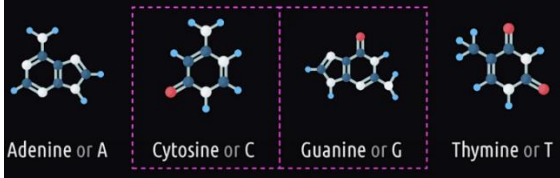
Example1:

```
my_dic = {
   "Spider" : "photographer",
   "Bat": "philantropist",
   "Wonder Wo": "nurse"
}

#1.open curely brackets in which our comprehension will take place
#2. replace each our keys with keys of man
#3.assign it to val
#4.use it to all the list
#5.call the items otherwise you will only be fetching the keys

#below will add man and remoe spiderman and replace with superman and superman is journalist so will replace that instead of photographer
my_dict = {(key+ "man" if key!= "Spider" else "Superman"):
          (val if val !="photographer" else "journalist") for (key,val) in my_dic.items()}
print(my_dict)# {'Superman': 'photographer', 'Batman': 'philantropist', 'Wonder Woman': 'nurse'}
```

Example2:

Real-life EXAMPLE

There are 4 nucleobases in DNA:



- A can only be paired with T
- C can only be paired with G

Real-life EXAMPLE

BASES

- Adenine
- Thymine
- Cytosine
- Guanine

Real-life EXAMPLE

BASES

- Adenine
- Thymine
- Cytosine
- Guanine

```
dna = {}

for idx, b in enumerate(dna_st1):
    if b == 'A':
        b2 = 'T'
    elif b == 'T':
        b2 = 'A'
    elif b == 'C':
        b2 = 'G'
    else:
        b2 = 'A'
    dna[idx] = [b, b2]
```

Real-life EXAMPLE

EXERCISE

- create strand 1 (list) with random bases.
- create a pairable strand 2 (list) based on the values from strand 1.
- create a dictionary with sequential keys.
- assign each key to a pair of bases such that:

```
{0: [s1[0], s2[0]], ...}
```

BASES

A + T

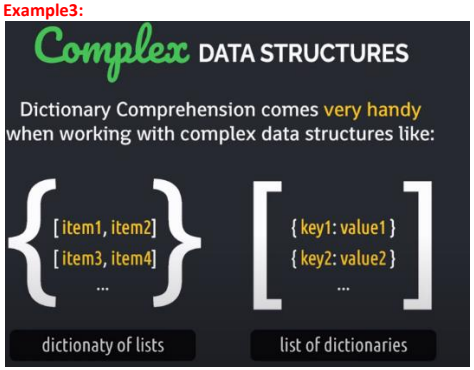
C + G

```
          s1    s2
{
    0:   ['A', 'T'],
    1:   ['C', 'G'],
    2:   ['C', 'G'],
    3:   ['A', 'T'],
    4:   ['C', 'G'],
    5:   ['G', 'C'],
    6:   ['T', 'A'],
    7:   ['C', 'G'],
    8:   ['C', 'G'],
    9:   ['A', 'T']
}
```

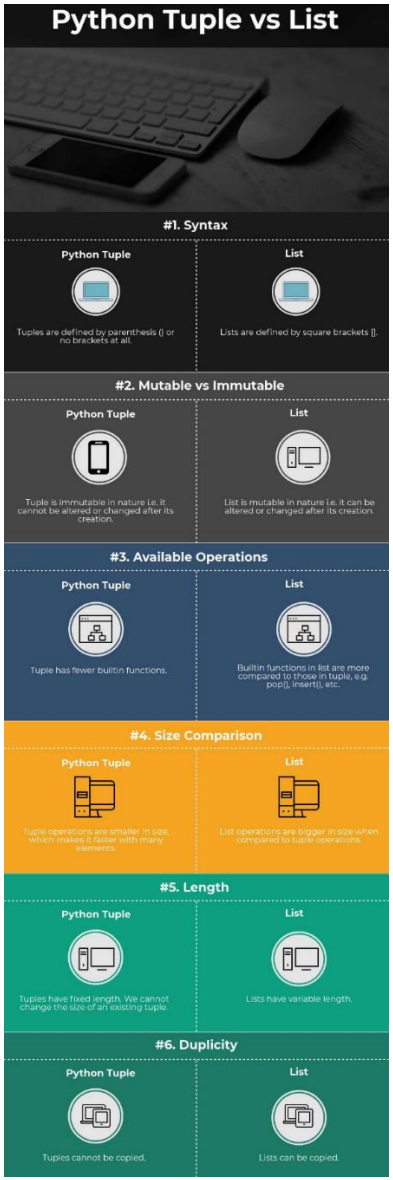
key value strand1 strand2

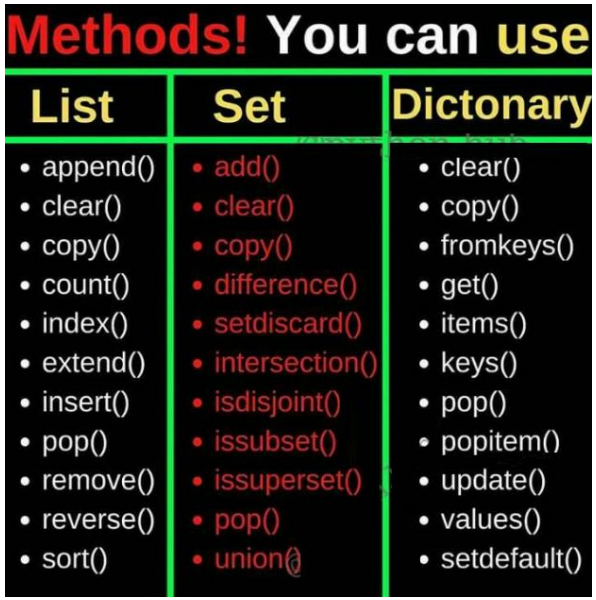
```
import random

bases = ["A","T","C","G"]

strand1 = random.choices(bases, k=10)
print(strand1)

dna = {key:[val,("T" if val == "A" else "A" if val == "T" else "C" if val =="G" else "G")]
          for (key, val) in enumerate(strand1)
      }
print(dna)
#{0: ['T', 'A'], 1: ['G', 'C'], 2: ['T', 'A'], 3: ['G', 'C'],
# 4: ['A', 'T'], 5: ['C', 'G'], 6: ['A', 'T'], 7: ['A', 'T'],
# 8: ['G', 'C'], 9: ['G', 'C']}
```

Example3:




```
import random
import string

# below will generate pass and id automatically and asign to user
keys = ["id", "username","password"]
users = ["mariyasha888","KnotABot","SpongiBob","iambatman"]

data = [{key:(i if key =="id" else users[i] if key == "username" else "".join(random.choices(string.printable, k=8))) for key in keys} for i in range(len(users))]

password= "".join(random.choices(string.printable, k=8)) #k is how long th passwotd will be
print(data)
#[{'id': 0, 'username': 'mariyasha888', 'password': '}Q`S#zqJ'},
# {'id': 1, 'username': 'KnotABot', 'password': 't=N43U.,'},
# {'id': 2, 'username': 'SpongiBob', 'password': 'Ql=p##z3'},
# {'id': 3, 'username': 'iambatman', 'password': '$@\n\x0bj!VY'}]
```





List Vs Set Vs Dictionary Vs Tuple

Lists	Sets	Dictionaries	Tuples
List = [10, 12, 15]	Set = {1, 23, 34} Print(set) -> {1, 23,24} Set = {1, 1} print(set) -> {1}	Dict = {"Ram": 26, "mary": 24}	Words = ("spam", "egss") Or Words = "spam", "eggs"
Access: print(list[0])	Print(set). Set elements can't be indexed.	print(dict["ram"])	Print(words[0])
Can contains duplicate elements	Can't contain duplicate elements. Faster compared to Lists	Can't contain duplicate keys, but can contain duplicate values	Can contains duplicate elements. Faster compared to Lists
List[0] = 100	set.add(7)	Dict["Ram"] = 27	Words[0] = "care" ->TypeError
Mutable	Mutable	Mutable	Immutable - Values can't be changed once assigned
List = []	Set = set()	Dict = {}	Words = ()
Slicing can be done print(list[1:2]) -> [12]	Slicing: Not done.	Slicing: Not done	Slicing can also be done on tuples
Usage: Use lists if you have a collection of data that doesn't need random access. Use lists when you need a simple, iterable collection that is modified frequently.	Usage: - Membership testing and the elimination of duplicate entries. - when you need uniqueness for the elements.	Usage: - When you need a logical association b/w key:value pair. - when you need fast lookup for your data, based on a custom key. - when your data is being constantly modified.	Usage: Use tuples when your data cannot change. A tuple is used in combination with a dictionary, for example, a tuple might represent a key, because its immutable.

C.6. Python Data Structures – OOP:

- **Excerpt from a 1994 Rolling Stone interview, Jobs explains what object-oriented programming is:**

Jeff Goodell:

Would you explain, in simple terms, exactly what object-oriented software is?

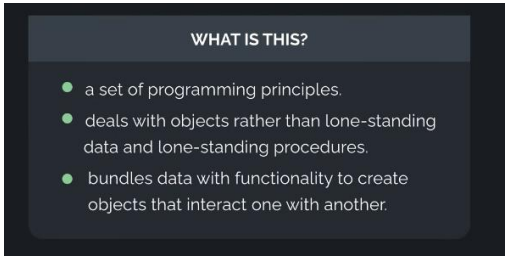
Steve Jobs:

Objects are like people. They're living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we're doing right here

Here's an example: If I'm your laundry object, you can give me your dirty clothes and send me a message that says, "Can you get my clothes laundered, please." I happen to know where the best laundry place in San Francisco is. And I speak English, and I have dollars in my pockets. So, I go out and hail a taxicab and tell the driver to take me to this place in San Francisco. I go get your clothes laundered, I jump back in the cab, I get back here. I give you your clean clothes and say, "Here are your clean clothes."

You have no idea how I did that. You have no knowledge of the laundry place. Maybe you speak French, and you can't even hail a taxi. You can't pay for one, you don't have dollars in your pocket. Yet I knew how to do all of that. And you didn't have to know any of it. All that complexity was hidden inside of me, and we were able to interact at a very high level of abstraction. That's what objects are. They encapsulate complexity, and the interfaces to that complexity are high level

So…



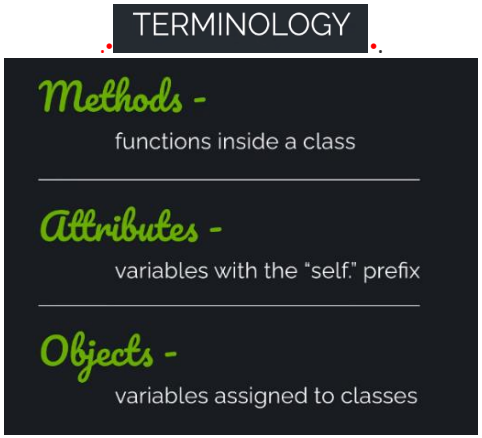
- **OOP** is a programming language model in which programs are organized around objects, rather than functions and logic
- Classes and Objects are core concepts of **OOPs**
- OOP is a thing that you want to store and process data about
- Class is a template for creating objects with related data and functions that they use the data
- Classes group data that we call **fields (attributes)** and related functions which we call **methods** into kind of factory in creating many objects that we call **instances**

- By adding features to class, our simple class then:

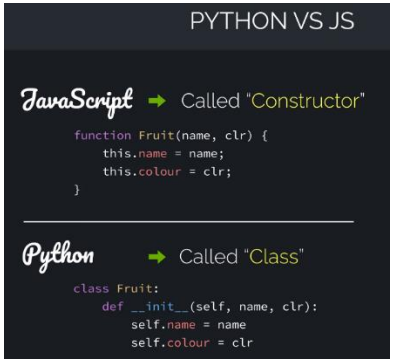
Will transforms into a DATA POWERHOUSE:

- Instead of the code below, it would be nicer to tell python that we want to write a datatype of our own that will allow us to write a code that we can re-use in the future if needed and each instance could have attributes to describe related information about it
- Meaning we can create random variables like below **but there will be NO relationship between them (even though you have written item1…, python takes it as a single variable with its designated type)**

```python
item1 = 'Phone'
item1_price = 100
item1_quantity = 5
item1_price_total = item1_price * item1_quantity

print(type(item1))  # str# <class 'str'>
print(type(item1_price))  # int # <class 'int'>
print(type(item1_quantity))  # int # <class 'int'>
print(type(item1_price_total))  # int # <class 'int'>

# meaning that those datatypes are instances of strings or integers
# in python each datatype is an object that has been instantiated earlier by some class
# for item1 variable, this has been instantiated from a string type of class
```



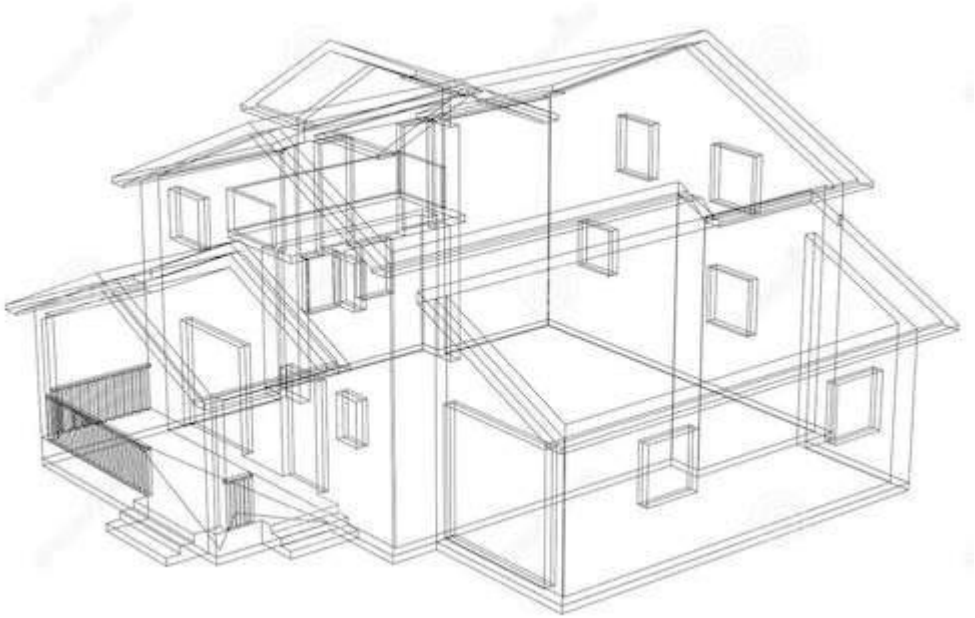
C.6.1. Classes:



- **A class is a template/blueprint for creating objects.** It provides: ***properties*** **(**what all data an object can have **— variables**)
  Behavior (things an object can do **— methods**)
- Class is a special data type
- Hold elements of many types
- To import class if it is in another module you write: from car import Car
- Class is a user-defined data structure that binds the data members and methods into a single unit

C.6.1.1. Analogy:

- **Think of a class as a blueprint of a house** — it's not a real physical house but just a specification with all the properties a house will have — rooms, floors, area etc.



House Class = Just a blueprint

C.6.1.2. Create a class:

- In Python, class is defined by using the `class` keyword. The syntax to create a class is given below:

```
class class_name:
  '''This is a Docstring. I have created a new class'''
  <statement 1>
  <statement 2>
  .
  .
  <statement N>
```

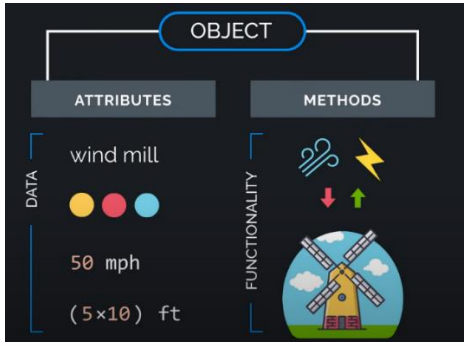
So:

- `class_name`: It is the name of the class
- `Docstring`: It is the first string inside the class and has a brief description of the class. Although not mandatory, this is highly recommended
  - **Sting inside triple quotes that you type right after the first line -> then you can call help(class_name) ->>>>> always use Docstring `help(class_name)`**
- `statements`: Attributes and methods

- **Things to remember about a Python class**
  - It is used to create custom types
  - The ***class*** keyword is used to define a new class
  - It defines the structure and behavior of objects
  - Conventionally a class name uses ***CamelCase***

```
⇨  # Class definition
   class House:
     pass
```

C.6.2. Objects:



- **Object: An object is an instance of a class**
- **An Object is a thing from the real world**
- **Another name for an object is 'entity'**
- Python is an Object-Oriented Programming language, so everything in Python is treated as an object. An object is a real-life entity. **It is the collection of various data and functions that operate on those data**
- Objects are real, have a state and occupy memory
- **We use the object of a class to perform actions**
- Two different objects of a class can have different values for properties and are (mostly) independent of each other (Taking the example of *House* class, if there are 2 objects of class House — they can have a different number of rooms, area, address, # of floors..)

- **When we say creating an instance or creating an object this means the same thing**
- **Created/instantiated from a class specification => cannot exist without a class**
- **In short, every object has the following property:**

Identity: Every object must be uniquely identified
State: An object has an **attribute** (variable) that represents a state of an object, and it also reflects the property of an object, **using its methods, we can modify its state**
Behavior: An object has **methods** (functions) that represent its behavior

Attributes and Methods of a Car

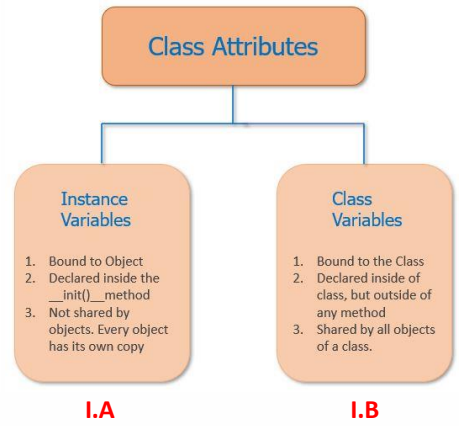
C.6.2.1. Analogy:

• **Think of an object as an actual physical house (with real brick and mortar) — created as per the blueprint (class house) defined above.**

A house = an instance/object of class House

C.6.2.2. Attributes & Methods:

I. Attributes:



I.A ▸ Instance Attributes (Instance Variables):

```
class Item:
    pass

item1 = Item()

item1.name = 'Phone' # by using '.' after the instance of a class we can create attributes, so we will give it an attribute that it is equal to 'Phone'
item1.price = 100
item1.quantity = 5

# so the difference between the random variables created earlier and now is that we have a relationship between those 4 lines
# each one of the attributes are assigned to one instance of the class
# proof: you can print item1 with types:
            # print(type(item1)) # item # <class '__main__.Item'>
           # print(type(item1.name)) # str # <class 'str'>
          # print(type(item1.price)) # int # <class 'int'>
         # print(type(item1.quantity)) #int # <class 'int'>
# so as per above this is how you create your own datatypes
```

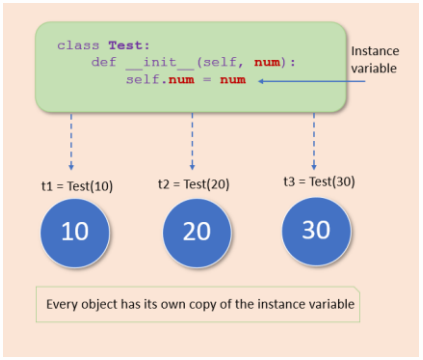
▸ **Above example is hard coding but you can use Magic Methods (__init__) to avoid this** ◂

- **There are several kinds of variables in Python:**
  - **Instance variables** in a class: these are called fields or attributes of an object
  - **Local Variables**: Variables in a method or block of code
  - **Parameters**: Variables in method declarations
  - **Class variables**: This variable is shared between all objects of a class in Object-oriented programming, when we design a class, we use instance variables and class variables
  - **Instance variables**: If the value of a variable varies from object to object, then such variables are called instance variables
  - **Class Variables**: A class variable is a variable that is declared inside of class, but outside of any instance method or `__init__()` method
- **When you create a variable inside a class it will be called attribute that allows to add data to an object**
- **The only difference between a variable and attribute is that an attribute is defined inside the class**

A.1. What is an Instance Variable in Python?

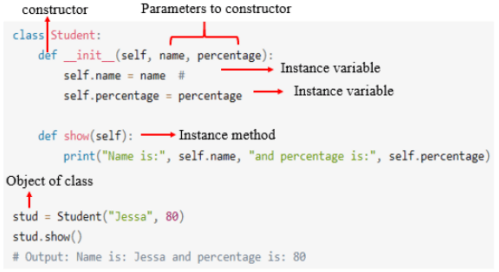
▸ **If the value of a variable varies from object to object, then such variables are called instance variables**. For every object, a separate copy of the instance variable will be created
▸ Instance variables are not shared by objects. Every object has its own copy of the instance attribute. This means that for each object of a class, the instance variable value is different
▸ When we create classes in Python, instance methods are used regularly. we need to create an object to execute the block of code or action defined in the instance method
▸ Instance variables are used within the instance method. We use the instance method to perform a set of actions on the data/value provided by the instance variable
▸ We can access the instance variable using the object and dot (.) operator

In Python, to work with an instance variable and method, we use the self keyword. We use the self keyword as the first parameter to a method. The self refers to the current object



Declare Instance variable

A.2. Create Instance Variables:



- **Instance Variables & Methods: Instance variables are declared inside a method using the self keyword, so when we pass the self parameter from method to a method (def__init__ TO def show), we can then access same attributes from __init__**
- **We use a constructor to define and initialize the instance variables**

Let's see the example below to declare an instance variable in Python. In the example, we are creating two instance variable name and age in the Student class:

```
class Student:
    # constructor
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

# create first object
s1 = Student("Jessa", 20)

# access instance variable
print('Object 1')
print('Name:', s1.name)
print('Age:', s1.age)

# create second object
s2= Student("Kelly", 10)

# access instance variable
print('Object 2')
print('Name:', s2.name)
print('Age:', s2.age)
```

Output

```
Object 1
```

```
Name: Jessa
Age: 20

Object 2
Name: Kelly
Age: 10
```

Note:

- When we created an object, we passed the values to the instance variables using a constructor
- Each object contains different values because we passed different values to a constructor to initialize the object
- Variable declared outside __init__() belong to the class. They're shared by all instances

```
class House:

  # Initializer
  def __init__(self, num_rooms):
    self.rooms = num_rooms

# Create instances of the class
house_small = House(2)
house_big = House(5)

print(f'# of rooms in small house: {house_small.rooms}')
print(f'# of rooms in big house: {house_big.rooms}')

Output:
# of rooms in small house: 2
# of rooms in big house: 5
```

A.3. Modify Values of Instance Variables:

- We can modify the value of the instance variable and assign a new value to it using the **object reference**

Note: When you change the instance variable's values of one object, the changes will not be reflected in the remaining objects because every object maintains a separate copy of the instance variable

Example:

```
class Student:
  # constructor
  def __init__(self, name, age):
    # Instance variable
    self.name = name
    self.age = age

# create object
stud = Student("Jessa", 20)

print('Before')
print('Name:', stud.name, 'Age:', stud.age)

# modify instance variable
stud.name = 'Emma'
stud.age = 15

print('After')
print('Name:', stud.name, 'Age:', stud.age)
```

Output

```
Before
Name: Jessa Age: 20

After
Name: Emma Age: 15
```

A.4. Ways to Access Instance Variable:

- **There are two ways to access the instance variable of class:**
  - **A.4.1. Within the class in instance method by using the object reference (self)**
  - **A.4.2. Using getattr() method**

A.4.1. Access instance variable in the instance method:

```
class Student:
    # constructor
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

    # instance method access instance variable
    def show(self):
        print('Name:', stud.name, 'Age:', stud.age)

# create object
stud = Student("Jessa", 20)

# call instance method
stud.show()
```

Output

```
Name: Jessa Age: 20
```

A.4.2. Access instance variable using getattr():

```
getattr(Object, 'instance_variable')
```

Pass the object reference and instance variable name to the getattr() method to get the value of an instance variable:

```
class Student:
    # constructor
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

# create object
stud = Student("Jessa", 20)

# Use getattr instead of stud.name
print('Name:', getattr(stud, 'name'))
print('Age:', getattr(stud, 'age'))
```

Output

```
Name: Jessa
Age: 20
```

A.5. Instance Variables Naming Conventions:

- Instance variable names should be all lower case. For example, `id`
- Words in an instance variable name should be separated by an underscore. For example, `store_name`
- Non-public instance variables should begin with a single underscore

- If an instance name needs to be mangled, two underscores may begin its name

A.6. Dynamically Add Instance Variable to a Object:

- We can add instance variables from the outside of class to a particular object. Use the following syntax to add the new instance variable to the object

```
object_referance.variable_name = value
```

Example:

```
class Student:
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

# create object
stud = Student("Jessa", 20)

print('Before')
print('Name:', stud.name, 'Age:', stud.age)

# add new instance variable 'marks' to stud
stud.marks = 75
print('After')
print('Name:', stud.name, 'Age:', stud.age, 'Marks:', stud.marks)
```

Output

```
Before
Name: Jessa Age: 20

After
Name: Jessa Age: 20 Marks: 75
```

Note:

- We cannot add an instance variable to a class from outside because instance variables belong to objects
- **Adding an instance variable to one object will not be reflected the remaining objects** because every object has a separate copy of the instance variable

A.7. Dynamically Delete Instance Variable:

- In Python, we use the del statement and delattr() function to delete the attribute of an object. Both of them do the same thing

A.7.1. del statement: The del keyword is used to delete objects. In Python, everything is an object, so the del keyword can also be used to delete variables, lists, or parts of a list, etc

A.7.2. delattr() function: Used to delete an instance variable dynamically

Note: When we try to access the deleted attribute, it raises an attribute error

A.7.1. Using the del statement:

```
class Student:
    def __init__(self, roll_no, name):
        # Instance variable
        self.roll_no = roll_no
        self.name = name

# create object
s1 = Student(10, 'Jessa')
print(s1.roll_no, s1.name)

# del name
del s1.name
# Try to access name variable
print(s1.name)
```

Output

```
10 Jessa
AttributeError: 'Student' object has no attribute 'name'
```

A.7.2. delattr() function:

- The delattr() function is used to delete the named attribute from the object with the prior permission of the object. Use the following syntax:

```
delattr(object, name)
```

- **object**: the object whose attribute we want to delete
- **name**: the name of the instance variable we want to delete from the object

Example:

```
class Student:
    def __init__(self, roll_no, name):
        # Instance variable
        self.roll_no = roll_no
        self.name = name

    def show(self):
        print(self.roll_no, self.name)

s1 = Student(10, 'Jessa')
s1.show()

# delete instance variable using delattr()
delattr(s1, 'roll_no')
s1.show()
```

Output

```
10 Jessa
AttributeError: 'Student' object has no attribute 'roll_no'
```

A.8. Access Instance Variable From Another Class:

- We can access instance variables of one class from another class using object reference. It is useful when we implement the concept of inheritance in Python, and we want to access the parent class instance variable from a child class
- let's understand this with the help of an example
- In the below example, the engine is an instance variable of the Vehicle class. We inherited a Vehicle class to access its instance variables in Car class

```
class Vehicle:
    def __init__(self):
        self.engine = '1500cc'

class Car(Vehicle):
    def __init__(self, max_speed):
        # call parent class constructor || come here ||
        super().__init__()
        self.max_speed = max_speed

    def display(self):
        # access parent class instance variables 'engine'
        print("Engine:", self.engine)
        print("Max Speed:", self.max_speed)

# Object of car
car = Car(240)
car.display()
```

Output

```
Engine: 1500cc
Max Speed: 240
```

A.9. List all Instance Variables of a Object:

- We can get the list of all the instance variables the object has. Use the `__dict__` function of an object to get all instance variables along with their value
- The `__dict__` function returns a dictionary that contains variable name as a key and variable value as a value
- `__dict__` is a **magic attribute, takes all the attributes and convert it to a dictionary**
- **can be used for debugging reasons**

Syntax:

```
print(Item.__dict__)  # all the attributes for class level
print(item1.__dict__)  # all the attributes for instance level
```

Output:

{'__module__': '__main__', 'pay_rate': 0.8, '__init__': <function Item.__init__ at 0x0000022D0AE06A70>, 'calculate_total_price': <function Item.calculate_total_price at 0x0000022D0B258700>, '__dict__': <attribute '__dict__' of 'Item' objects>, '__weakref__': <attribute '__weakref__' of 'Item' objects>, '__doc__': None}

{'name': 'Phone', 'price': 100, 'quantity': 1}

Example:

```
class Student:
    def __init__(self, roll_no, name):
        # Instance variable
        self.roll_no = roll_no
        self.name = name

s1 = Student(10, 'Jessa')
print('Instance variable object has')
print(s1.__dict__)

# Get each instance variable
for key_value in s1.__dict__.items():
    print(key_value[0], '=', key_value[1])
```

Output:

```
Instance variable object has
{'roll_no': 10, 'name': 'Jessa'}

roll_no = 10
name = Jessa
```

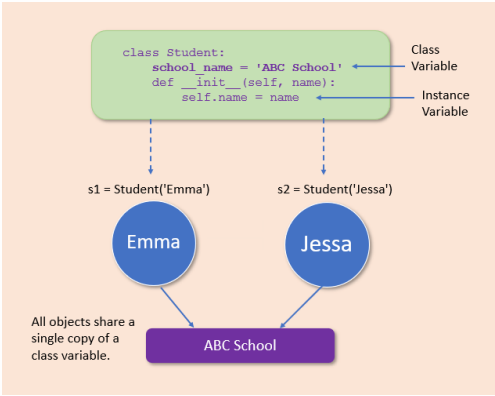
I.B Class Attributes (Class variables, Static Variables):

```
# class attribute is a class that belongs to the class itself
# you can access class attribute also from the instance itself
# class attribute will be global or across all instances
# example if you want to apply sale on your shop so you have to apply discount on each of your item so the class variable will be shared across all instances
# we can access the class attribute from the instance level, ex. print(Item.pay_rate)

class Item():
    pay_rate  = 0.8 # class attribute – class level # The pay rate after 20% discount, pay_rate can either be accessed from class level or instance level

    def __init__(self, name: str, price: float, quantity=0):
        assert price >= 0, f"Price {price} is not greater or equal to zero"
        assert quantity >= 0, f"Price {quantity} is not greater or equal to zero"

        # assign to self object # instance level #
        self.name = name
        self.price = price
        self.quantity = quantity
```

```
    def calculate_total_price(self):
        return self.price * self.quantity

    def apply_discount(self):
        self.price = self.price * self.pay_rate # this will be read pay_rate from Item level (class level) so we add self


item1 = Item("Phone" , 100, 1)

print(Item.pay_rate)  # output: 0.8
print(item1.pay_rate) # output: 0.8, so item1 tried to bring this attribute from the instance level (where we have self.name etc.) but as it could not find pay_rate attribute there it went to the class attribute and found it there and brought it

item1.apply_discount()
print(item1.price) # output 80.0

# if you want to apply different discount to a specific item
# so you will assign the same attribute to the instance
item2 = Item("Laptop", 1000, 3)
item2.pay_rate = 0.7 # so will find the attribute of pay_rate at the instance level so does not have to go to the class level to bring the value of pay_rate because it is going to find it at the instance level which actually is  item2.pay_rate = 0.7,
                     basically right here where we gave the attribute 0.7 value, so if we overwrite the pay_rate of the instance for the instance level then it will read from the instance level facing item1 that it will pull the value from the class level
item2.apply_discount()
print(item2.price) # output 700
```

B.1. What is an Class Variable in Python?

- If the **value of a variable is not varied from object to object**, such types of variables are called **class variables or static variables**
- Class variables are **shared by all instances of a class**. Unlike instance variable, the value of a class variable is not varied from object to object
- In Python, Class variables are declared when a class is being constructed
  -> They are not defined inside any methods of a class because of this only one copy of the static variable will be created, and shared between all objects of the class
- For example, in Student class, we can have different instance variables such as name and roll number because each student's name and roll number are different
  -> But, if we want to include the school name in the student class, we must use the class variable instead of an instance variable as the school name is the same for all students
    -> So instead of maintaining the separate copy in each object, we can create a class variable that will hold the school name so all students (objects) can share it
- We can add any number of class variables in a class



Understand Class Variables

B.2. Create Class Variables:

- A class variable is declared inside of class, but outside of any instance method or __init__() method
- By convention, typically it is placed right below the class header and before the constructor method and other methods

Example:

```
class Student:
    # Class variable
    school_name = 'ABC School '

    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no
```

```
# create first object
s1 = Student('Emma', 10)
print(s1.name, s1.roll_no, Student.school_name)
# access class variable

# create second object
s2 = Student('Jessa', 20)
# access class variable
print(s2.name, s2.roll_no, Student.school_name)
```

Output

```
Emma 10 ABC School
Jessa 20 ABC School
```

- In the above example, we created the class variable school_name and accessed it using the object and class name

  Note: **Like regular variables, class variables can store data of any type. We can use Python list, Python tuple, and Python dictionary as a class variable**

B.3. Accessing Class Variables:

- We can access static variables either by class name or by object reference, but it is recommended to use the class name
- In Python, we can access the class variable in the following places:

B.3.1. Access inside the constructor by using either self parameter or class name

B.3.2. Access class variable inside instance method by using either self of class name

B.3.3. Access from outside of class by using either object reference or class name

B.3.1. Access Class Variable in the constructor:

```
class Student:
    # Class variable
    school_name = 'ABC School '

    # constructor
    def __init__(self, name):
        self.name = name
        # access class variable inside constructor using self
        print(self.school_name)
        # access using class name
        print(Student.school_name)

# create Object
s1 = Student('Emma')
```

Output

```
ABC School
ABC School
```

B.3.2. Access Class Variable in Instance method and outside class:

```
class Student:
    # Class variable
    school_name = 'ABC School '

    # constructor
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

    # Instance method
    def show(self):
        print('Inside instance method')
        # access using self
```

```
        print(self.name, self.roll_no, self.school_name)
        # access using class name
        print(Student.school_name)

# create Object
s1 = Student('Emma', 10)
s1.show()

print('Outside class')
# access class variable outside class
# access using object reference
print(s1.school_name)

# access using class name
print(Student.school_name)
```

Output

```
Inside instance method
Emma 10 ABC School
ABC School

Outside class
ABC School
ABC School
```

- In above example, we accessed the class variable `school_name` using class name and a `self` keyword inside a method

B.4. Modify Class Variables:

- Generally, we assign value to a class variable inside the class declaration. However, we can change the value of the class variable either in the class or outside of class

Note: We should change the class variable's value using the class name only

Example:

```
class Student:
    # Class variable
    school_name = 'ABC School '

    # constructor
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

    # Instance method
    def show(self):
        print(self.name, self.roll_no, Student.school_name)

# create Object
s1 = Student('Emma', 10)
print('Before')
s1.show()

# Modify class variable
Student.school_name = 'XYZ School'
print('After')
s1.show()
```

Output:

```
Before
Emma 10 ABC School

After
```

```
Emma 10 XYZ School
```

Note:
It is best practice to use a class name to change the value of a class variable. Because if we try to change the class variable's value by using an object, a new instance variable is created for that particular object, which shadows the class variables.

Example:

```
class Student:
    # Class variable
    school_name = 'ABC School '

    # constructor
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

# create Objects
s1 = Student('Emma', 10)
s2 = Student('Jessa', 20)

print('Before')
print(s1.name, s1.roll_no, s1.school_name)
print(s2.name, s2.roll_no, s2.school_name)

# Modify class variable using object reference
s1.school_name = 'PQR School'
print('After')
print(s1.name, s1.roll_no, s1.school_name)
print(s2.name, s2.roll_no, s2.school_name)
```

Output:

```
Before
Emma 10 ABC School
Jessa 20 ABC School

After
Emma 10 PQR School
Jessa 20 ABC School
```

A new instance variable is created for the s1 object, and this variable shadows the class variables. So always use the class name to modify the class variable.

B.5. Class Variables In Inheritance:

- As you know, only one copy of the class variable will be created and shared between all objects of that class
- When we use inheritance, all variables and methods of the base class are available to the child class. In such cases, We can also change the value of the parent class's class variable in the child class
- We can use the parent class or child class name to change the value of a parent class's class variable in the child class

Example:

```
class Course:
    # class variable
    course = "Python"

class Student(Course):

    def __init__(self, name):
        self.name = name

    def show_student(self):
        # Accessing class variable of parent class
        print('Before')
```

```
        print("Student name:", self.name, "Course Name:", Student.course)
        # changing class variable value of base class
        print('Now')
        Student.course = "Machine Learning"
        print("Student name:", self.name, "Course Name:", Student.course)

# creating object of Student class
stud = Student("Emma")
stud.show_student()
```

Output

```
Before
Student name: Emma Course Name: Python

Now
Student name: Emma Course Name: Machine Learning
```

- What if both **child class and parent class has the same class variable name**. In this case, the child class will not inherit the class variable of a base class
- So, it is recommended to create a separate class variable for child class instead of inheriting the base class variable

Example:

```
class Course:
    # class variable
    course = "Python"

class Student(Course):
    # class variable
    course = "SQL"

    def __init__(self, name):
        self.name = name

    def show_student(self):
        # Accessing class variable
        print('Before')
        print("Student name:", self.name, "Course Name:", Student.course)
        # changing class variable's value
        print('Now')
        Student.course = "Machine Learning"
        print("Student name:", self.name, "Course Name:", Student.course)

# creating object of Student class
stud = Student("Emma")
stud.show_student()

# parent class course name
print('Parent Class Course Name:', Course.course)
```

Output:

```
Before
Student name: Emma Course Name: SQL
Now
Student name: Emma Course Name: Machine Learning

Parent Class Course Name: Python
```

B.6. Wrong Use of Class Variables:

- In Python, we should properly use the class variable because all objects share the same copy. Thus, if one of the objects modifies the value of a class variable, then all objects start referring to the fresh copy

Example:

```
class Player:
    # class variables
    club = 'Chelsea'
    sport = 'Football'

    def __init__(self, name):
        # Instance variable
        self.name = name

    def show(self):
        print("Player :", 'Name:', self.name, 'Club:', self.club, 'Sports:', self.sport)

p1 = Player('John')

# wrong use of class variable
p1.club = 'FC'
p1.show()

p2 = Player('Emma')
p2.sport = 'Tennis'
p2.show()

# actual class variable value
print('Club:', Player.club, 'Sport:', Player.sport)
```

Output

```
Player : Name: John Club: FC Sports: Football
Player : Name: Emma Club: Chelsea Sports: Tennis
Club: Chelsea Sport: Football
```

- In the above example, the instance variable `name` is unique for each player. The class variable `team` and `sport` can be accessed and modified by any object
- Because both objects modified the class variable, a new instance variable is created for that particular object with the same name as the class variable, which shadows the class variables
- In our case, for object `p1` new instance variable `club` gets created, and for object `p2` new instance variable `sport` gets created
- So when you try to access the class variable using the `p1` or `p2` object, it will not return the actual class variable value
- To avoid this, always modify the class variable value using the class name so that all objects gets the updated value. Like this:

```
Player.club = 'FC'
Player.sport = 'Tennis'
```

Class Variable vs Instance variables:

- The following table shows the difference between the instance variable and the class variable
- In Python, properties can be defined into two parts:
  - **Instance variables**: Instance variable's value varies from object to object. Instance variables are not shared by objects. Every object has its own copy of the instance attribute
  - **Class Variables**: A class variable is a variable that is declared inside of class, but outside of any instance method or __init__() method. Class variables are shared by all instances of a class

	Instance Variable	Class Variable
	Instance variables are not shared by objects. Every object has its own copy of the instance attribute	Class variables are shared by all instances.
	Instance variables are declared inside the constructor i.e., the __init__() method.	Class variables are declared inside the class definition but outside any of the instance methods and constructors.
	It is gets created when an instance of the class is created.	It is created when the program begins to execute.
	Changes made to these variables through one object will not reflect in another object.	Changes made in the class variable will reflect in all objects.

Class Variables vs. Instance Variables

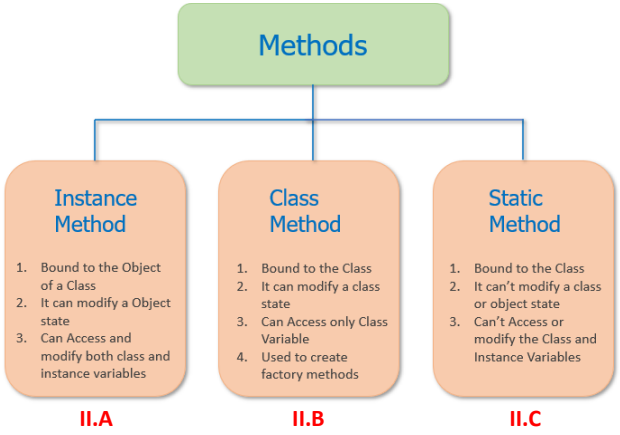
Example:
Let's see the below example to create a class variable and instance variable:

```
class Car:
    # Class variable
    manufacturer = 'BMW'

    def __init__(self, model, price):
        # instance variable
        self.model = model
        self.price = price

# create Object
car = Car('x1', 2500)
print(car.model, car.price, Car.manufacturer)
```

Output:

```
x1 2500 BMW
```

II. Methods:



- In Object-oriented programming, we use instance methods and class methods. Inside a Class, we can define the following three types of methods:
  - **A. Instance method**: Used to access or modify the object state. If we use instance variables inside a method, such methods are called instance methods. It must have a self parameter to refer to the current object
  - **B. Class method**: Used to access or modify the class state. In method implementation, if we use only class variables, then such type of methods we should declare as a class method
    - The class method has a cls parameter which refers to the class
  - **C. Static method**: It is a general utility method that performs a task in isolation. Inside this method, we don't use instance or class variable because this static method doesn't take any parameters like self and cls

II.A ▸ Instance Method (Instance function):

```
class Item():
    def calculate_total_price(self, x, y):
        return x * y

# we could go inside our class and write methods that will be accessible from our instances
# when we say methods it actually means functions inside classes
# python passes the object as the its first argument every time when calling those methods
# so you need to receive at least 1 parameter when creating methods
# self can be any word but the convention is "self"

item1 = Item()

item1.name = 'Phone'
item1.price = 100
item1.quantity = 5

print(item1.calculate_total_price(item1.price, item1.quantity))
# so python will pass item1 as the first argument 'self'
# then python will pass item1.price as the second argument 'x'
# python then will pass item1.quantity as the third argument 'y'

item2 = Item()
item2.name = 'Laptop'
item2.price = 1000
item2.quantity = 3

print(item2.calculate_total_price(item2.price, item2.quantity))
```

⇨

```
class House:
    # define instance method
    def set_rooms(self, num_rooms):
        self.rooms = num_rooms

            # Create instance of the class - creating an actual house
             house_small = House()
```

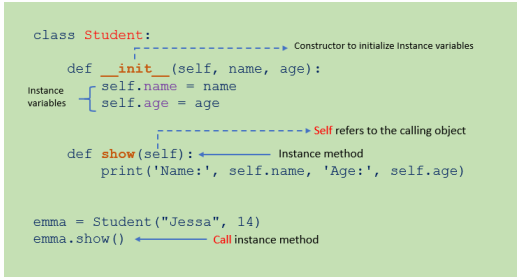
```
# Calling instance method for instance 'house_small'
house_small.set_rooms(2)

# Create another instance
house_big = House()
house_big.set_rooms(5)

print(f'# of rooms in small house: {house_small.rooms}')
print(f'# of rooms in big house: {house_big.rooms}')

Output:
# of rooms in small house: 2
# of rooms in big house: 5
```

recipe (how to make the cookie) vs cookie
cookie = house_small = instance
↥ the recipe =House() = class = the class definition ↧

- **Things to remember about Instance methods**
  - Called using objects
  - Must have ***self*** as the first parameter
  - (***self*** is another python term. We can use self to access any data or other instance methods which resides in that class. These cannot be accessed without using ***self***
    - **Thought for efficiency:** Doesn't it makes sense to specify the number of rooms in a house while creating the House object itself instead of doing it later?
      -> Constructor (instead of hard coding)

A.1. What is Instance Methods in Python?

- If we use instance variables inside a method, such methods are called instance methods. **The instance method performs a set of actions on the data/value provided by the instance variables**
- A instance method is bound to the object of the class
- It can access or modify the **object state** by changing the value of a instance variables
- When we create a class in Python, instance methods are used regularly
- To work with an instance method, we use the self keyword. We use the self keyword as the first parameter to a method. The self refers to the current object
- Any method we create in a class will automatically be created as an instance method unless we explicitly tell Python that it is a class or static method



Instance method in Python

A.2. Define Instance Method:

- Instance variables are not shared between objects. Instead, every object has its copy of the instance attribute. Using the instance method, we can access or modify the calling object's attributes
- Instance methods are defined inside a class, and it is pretty similar to defining a regular function
- Use the def keyword to define an instance method in Python
- **Use self as the first parameter in the instance method when defining it**. The self parameter refers to the current object
- Using the self parameter to access or modify the current object attributes
- You may use a variable named differently for self, but it is discouraged since self is the recommended convention in Python
- Let's see below example to create an instance method show() in the Student class to display the student details:

Example:

```
class Student:
  # constructor
  def __init__(self, name, age):
    # Instance variable
    self.name = name
    self.age = age

  # instance method to access instance variable
  def show(self):
```

```
        print('Name:', self.name, 'Age:', self.age)
```

A.3. Calling An Instance Method:

- We use an object and dot (.) operator to execute the block of code or action defined in the instance method
  - First, create instance variables name and age in the Student class
  - Next, create an instance method `display()` to print student name and age
  - Next, create object of a Student class to call the instance method
    - **Example** below is how to call an instance method show() to access the student object details such as name and age:

```
class Student:
    # constructor
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

    # instance method access instance variable
    def show(self):
        print('Name:', self.name, 'Age:', self.age)

# create first object
print('First Student')
emma = Student("Jessie", 14)
# call instance method
emma.show()

# create second object
print('Second Student')
kelly = Student("Kelly", 16)
# call instance method
kelly.show()
```

Output:

```
First Student
Name: Jessa Age: 14

Second Student
Name: Kelly Age: 16
```

Note:

- Inside any instance method, we can use self to access any data or method that reside in our class. We are unable to access it without a self parameter so self will act as a key when passed from __init__ it will allow to create the magic, which also def __init__ is actually a method
- An instance method can freely access attributes and even modify the value of attributes of an object by using the self parameter
- By Using self.__class__ attribute we can access the class attributes and change the class state. Therefore instance method gives us control of changing the object as well as the class state

A.4. Modify Instance Variables inside Instance Method:

- Let's create the instance method update() method to modify the student age and roll number when student data details change:

```
class Student:
    def __init__(self, roll_no, name, age):
        # Instance variable
        self.roll_no = roll_no
        self.name = name
        self.age = age

    # instance method access instance variable
```

```
    def show(self):
        print('Roll Number:', self.roll_no, 'Name:', self.name, 'Age:', self.age)

    # instance method to modify instance variable
    def update(self, roll_number, age):
        self.roll_no = roll_number
        self.age = age

# create object
print('class VIII')
stud = Student(20, "Emma", 14)
# call instance method
stud.show()

# Modify instance variables
print('class IX')
stud.update(35, 15)
stud.show()
```

Output:

```
class VIII
Roll Number: 20 Name: Emma Age: 14
class IX
Roll Number: 35 Name: Emma Age: 15
```

A.5. Create Instance Variables in Instance Method:

- Till the time we used constructor to create instance attributes. But, instance attributes are not specific only to the `__init__()` method; they can be defined elsewhere in the class
- So, let's see how to create an instance variable inside the method

Example:

```
class Student:
    def __init__(self, roll_no, name, age):
        # Instance variable
        self.roll_no = roll_no
        self.name = name
        self.age = age

    # instance method to add instance variable
    def add_marks(self, marks):
        # add new attribute to current object
        self.marks = marks

# create object
stud = Student(20, "Emma", 14)
# call instance method
stud.add_marks(75)

# display object
print('Roll Number:', stud.roll_no, 'Name:', stud.name, 'Age:', stud.age, 'Marks:', stud.marks)
```

Output:

```
Roll Number: 20 Name: Emma Age: 14 Marks: 75
```

A.6. Dynamically Add Instance Method to a Object:

- Usually, we add methods to a class body when defining a class. However, Python is a dynamic language that allows us to add or delete instance methods at runtime. Therefore, it is helpful in the following scenarios
  - When class is in a different file, and you don't have access to modify the class structure
  - You wanted to extend the class functionality without changing its basic structure because many systems use the same structure

  Let's see how to add an instance method in the Student class at runtime

Example:

- We should add a method to the object, so other instances don't have access to that method. We use the types module's `MethodType()` to add a method to an object. Below is the simplest way to method to an object:

```
import types

class Student:
    # constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def show(self):
        print('Name:', self.name, 'Age:', self.age)

# create new method
def welcome(self):
    print("Hello", self.name, "Welcome to Class IX")


# create object
s1 = Student("Jessa", 15)

# Add instance method to object
s1.welcome = types.MethodType(welcome, s1)
s1.show()

# call newly added method
s1.welcome()
```

Output:

```
Name: Jessa Age: 15
Hello Jessa Welcome to Class IX
```

A.7. Dynamically Delete Instance Methods:

- We can dynamically delete the instance method from the class. In Python, there are two ways to delete method:

i) By using the del operator
ii) By using delattr() method

i) By using the del operator:

- The del operator removes the instance method added by class.

Example:

- In below example, we will delete an instance method named `percentage()` from a Student class. If you try to access it after removing it, you'll get an Attribute Error:

```
class Student:
    # constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def show(self):
        print('Name:', self.name, 'Age:', self.age)

    # instance method
    def percentage(self, sub1, sub2):
        print('Percentage:', (sub1 + sub2) / 2)

emma = Student('Emma', 14)
emma.show()
emma.percentage(67, 62)

# Delete the method from class using del operator
del emma.percentage
```

```
# Again calling percentage() method
# It will raise error
emma.percentage(67, 62)
```

Output:

```
Name: Emma Age: 14
Percentage: 64.5

 File "/demos/oop/delete_method.py", line 21, in <module>
   del emma.percentage
AttributeError: percentage
```

ii) By using the delattr() method:

- The delattr() is used to delete the named attribute from the object with the prior permission of the object. Use the following syntax to delete the instance method:

```
delattr(object, name)
```

- **object**: the object whose attribute we want to delete
- **name**: the name of the instance method you want to delete from the object

Example:

- In below example, we will delete an instance method named percentage() from a Student class:

```
emma = Student('Emma', 14)
emma.show()
emma.percentage(67, 62)

# delete instance method percentage() using delattr()
delattr(emma, 'percentage')
emma.show()

# Again calling percentage() method
# It will raise error
emma.percentage(67, 62)
```

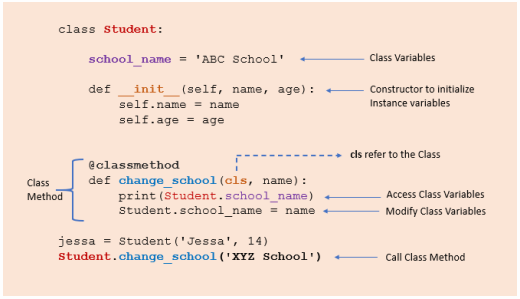
II.B ▸ Class Methods:

```
import csv

class Item():
    pay_rate  = 0.8 # class attribute # The pay rate after 20% discount
    all = [] # we will add or instances here

    def __init__(self, name: str, price: float, quantity=0):
        assert price >= 0, f"Price {price} is not greater or equal to zero"
        assert quantity >= 0, f"Price {quantity} is not greater or equal to zero"

        # assign to self object
        self.name = name
        self.price = price
        self.quantity = quantity

        # Actions to execute
        Item.all.append(self) # automatically the instances will fill our empty list

    def calculate_total_price(self):
        return self.price * self.quantity
```

```
    def apply_discount(self):
        self.price = self.price * self.pay_rate # self.pay_rate it has self in front because pay_rate can either be accessed from class level or instance level, therefore this will read from Item level (class level)

    def __repr__(self): # you have to write print(Item.all) -> or use a for loop->>> for instances in Item.all: print(instances.name)
        return f"Item('{self.name}',{self.price},{self.quantity})" # so write it like you are writing when you instantiated the objects

    # decorators in python are a quick way to change the behavior of the functions that we will write by calling them just before the line that we create our function
    # class method is a method that could be accessed from the class level only
    # we will not have any instances on our hand to call this method from the instance because this method is designed for instantiating the object itself
    # so this means that this method could not be called from an instance
    # by converting this method into a class method that will be the solution to be accessed from class level only
    @classmethod
    def instantiate_from_csv(cls): # the method will receive a parameter named 'cls', so when we call our class methods then the class object itself is passed as a first argument always in the background,
                                   # it's a bit like the instance where it is also passed as a first argument but this time when we call a class method in this approach then the class reference must be passed as a first argument,
                                   # and that is why you should still receive 1 parameter but not 'self' as it is going to be confusing
        # we will use context manager to read the items.csv file
        with open('items.csv','r') as f:
            reader = csv.DictReader(f) # so it will read our content as a list of dictionaries
            items = list(reader) # we converted the reader into a list

        for item in items:
            print(item) # it will print items like dictionaries
            Item(
                name=item.get('name'),
                price=float(item.get('price')),
                quantity=int(item.get('quantity')),
            )

    # csv - comma separated values
    # so we can put the above in a csv file
    # for the above will be great as it will be saved in a data structured format
    # in pycharm go to name of the project ->right click ->new->file-> name.csv

    Item.instantiate_from_csv() # this method should take full responsibility to instantiating those objects for us
    print(Item.all)
```

a) What is Class Method in Python?

- Class methods are methods that are called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method
  - **A class method is bound to the class** and not the object of the class. It can access only class variables
  - It can modify the class state by changing the value of a class variable that would apply across all the class objects
- In method implementation, if we use only class variables, we should declare such methods as class methods. The class method has a `cls` as the first parameter, which refers to the class
- Class methods are used when we are **dealing with factory methods**. Factory methods are those methods that **return a class object for different use cases**
  ->Thus, factory methods create concrete implementations of a common interface
- The class method can be called using `ClassName.method_name()` as well as by using an object of the class



b) Define Class Method:

- Any method we create in a class will automatically be created as an instance method
- We must explicitly tell Python that it is a class method using the `@classmethod` decorator or `classmethod()` function
- Class methods are defined inside a class, and it is pretty similar to defining a regular function
- Like, inside an instance method, we use the `self` keyword to access or modify the instance variables. Same inside the class method, we use the `cls` keyword as a first parameter to access class variables

-> Therefore the class method gives us control of changing the class state

- You may use a variable named differently for `cls`, but it is discouraged since self is the recommended convention in Python
- The class method can only access the class attributes, not the instance attributes

Example 1: Create Class Method Using @classmethod Decorator:

- To make a method as class method, add `@classmethod` decorator before the method definition, and add `cls` as the first parameter to the method
- The `@classmethod` decorator is a built-in function decorator
- In Python, we use the `@classmethod` decorator to declare a method as a class method
- The `@classmethod` decorator is an expression that gets evaluated after our function is defined
- Let's see how to create a factory method using the class method. In below example, we will create a Student class object using the class method:

```
from datetime import date

class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def calculate_age(cls, name, birth_year):
        # calculate age and set it as a age
        # return new object
        return cls(name, date.today().year - birth_year)

    def show(self):
        print(self.name + "'s age is: " + str(self.age))

jessa = Student('Jessa', 20)
jessa.show()

# create new object using the factory method
joy = Student.calculate_age("Joy", 1995)
joy.show()
```

Output

```
Jessa's age is: 20
John's age is: 26
```

- In the above example, we created two objects, one using the constructor and the second using the `calculate_age()` method
- The **constructor** takes two arguments name and age. On the other hand, class method takes `cls`, `name`, and `birth_year` and returns a class instance which is nothing but a new object
- The `@classmethod` decorator is used for converting `calculate_age()` method to a class method
- The `calculate_age()` method takes Student class (`cls`) as a first parameter and returns constructor by calling `Student(name, date.today().year - birthYear)`, which is equivalent to `Student(name, age)`

Example 2: Create Class Method Using classmethod() function:

- Apart from a decorator, the built-in function `classmethod()` is used to convert a normal method into a class method
- The **`classmethod()`** is an inbuilt function in Python, which returns a class method for a given function

Syntax:

```
classmethod(function)
```

- `function`: It is the name of the method you want to convert as a class method.
- It returns the converted class method.

Note: The method you want to convert as a class method must accept class (`cls`) as the first argument, just like an instance method receives the instance (`self`)

As we know, the class method is bound to class rather than an object. So we can call the class method both by calling class and object

A `classmethod()` function is the older way to create the class method in Python. In a newer version of Python, we should use the `@classmethod` decorator to create a class method

Example: Create class method using `classmethod()` function

```
class School:
    # class variable
```

```
    name = 'ABC School'

    def school_name(cls):
        print('School Name is :', cls.name)

# create class method
School.school_name = classmethod(School.school_name)

# call class method
School.school_name()
```

Output

```
School Name is : ABC School
```

Example 3: Access Class Variables in Class Methods:

- Using the class method, we can only access or modify the class variables. Let's see how to access and modify the class variables in the class method
- Class variables are **shared by all instances of a class**
- Using the class method, we can modify the class state by changing the value of a class variable that would apply across all the class objects

```
class Student:
    school_name = 'ABC School'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def change_school(cls, school_name):
        # class_name.class_variable
        cls.school_name = school_name

    # instance method
    def show(self):
        print(self.name, self.age, 'School:', Student.school_name)

jessa = Student('Jessa', 20)
jessa.show()

# change school_name
Student.change_school('XYZ School')
jessa.show()
```

Output:

```
Jessa 20 School: ABC School
Jessa 20 School: XYZ School
```

c) Class Method in Inheritance:

- In inheritance, the class method of a parent class is available to a child class
- Let's create a Vehicle class that contains a factory class method from_price() that will return a Vehicle instance from a price
- When we call the same method using the child's class name, it will return the child's class object
- Whenever we derive a class from a parent class that has a class method then it creates the correct instance of the derived class
- Below example shows how the class method works in inheritance:

Example:

```
class Vehicle:
    brand_name = 'BMW'

    def __init__(self, name, price):
        self.name = name
```

```
            self.price = price

        @classmethod
        def from_price(cls, name, price):
            # ind_price = dollar * 76
            # create new Vehicle object
            return cls(name, (price * 75))

        def show(self):
            print(self.name, self.price)

    class Car(Vehicle):
        def average(self, distance, fuel_used):
            mileage = distance / fuel_used
            print(self.name, 'Mileage', mileage)

    bmw_us = Car('BMW X5', 65000)
    bmw_us.show()

    # class method of parent class is available to child class
    # this will return the object of calling class
    bmw_ind = Car.from_price('BMW X5', 65000)
    bmw_ind.show()

    # check type
    print(type(bmw_ind))
```

Output

```
BMW X5 65000
BMW X5 4875000
class '__main__.Car'
```

d) Dynamically Add Class Method to a Class:

- Typically, we add class methods to a class body when defining a class
- However, Python is a dynamic language that allows us to add or delete methods at runtime
- Therefore, it is helpful when you wanted to extend the class functionality without changing its basic structure because many systems use the same structure
- We need to use the `classmethod()` function to add a new class method to a class

Example:

Let's see below how to add a new class method in the Student class at runtime:

```
class Student:
    school_name = 'ABC School'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def show(self):
        print(self.name, self.age)

# class ended

# method outside class
def exercises(cls):
    # access class variables
    print("Below exercises for", cls.school_name)

# Adding class method at runtime to class
Student.exercises = classmethod(exercises)

jessa = Student("Jessa", 14)
jessa.show()
# call the new method
```

```
Student.exercises()
```

Output

```
Jessa 14
Below exercises for ABC School
```

e) Dynamically Delete Class Methods:

- We can dynamically delete the class methods from the class. In Python, there are two ways to do it:

i) By using the `del` operator

ii) By using `delattr()` method

i) By using the `del` operator:

- The `del` operator removes the instance method added by class. Use the `del class_name.class_method` syntax to delete the class method

Example:

- In this example, we will delete the class method named `change_school()` from a Student class. If you try to access it after removing it, you'll get an Attribute Error

```
class Student:
    school_name = 'ABC School'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def change_school(cls, school_name):
        cls.school_name = school_name

jessa = Student('Jessa', 20)
print(Student.change_school('XYZ School'))
print(Student.school_name)

# delete class method
del Student.change_school

# call class method
# it will give error
print(Student.change_school('PQR School'))
```

Output

```
XYZ School
AttributeError: type object 'Student' has no attribute 'change_school'
```

ii) By using `delatt()` method:

- The `delattr()` method is used to delete the named attribute and method from the class. The argument to `delattr` is an object and string. The string must be the name of an attribute or method name

Example:

```
jessa = Student('Jessa', 20)
print(Student.change_school('XYZ School'))
print(Student.school_name)

# delete class method
delattr(Student, 'change_school')

# call class method
# it will give error
print(Student.change_school('PQR School'))
```

Output

```
XYZ School
AttributeError: type object 'Student' has no attribute 'change_school'
```

II.C ▸ Static Method:

C.1. What is Static Methods in Python:

- **A static method is a general utility method that performs a task in isolation**. Static methods in Python are similar to those found in Java or C++
- A static method is bound to the class and not the object of the class. Therefore, we can call it using the class name
- A static method doesn't have access to the class and instance variables because it does not receive an implicit first argument like self and cls. Therefore **it cannot modify the state of the object or class**
- The class method can be called using ClassName.method_name() as well as by using an object of the class

```
class Employee:
    @staticmethod
    def sample(x):
        print('Inside static method', x)

# call static method
Employee.sample(10)

# can be called using object
emp = Employee()
emp.sample(10)
```

C.2. Define Static Method in Python:

- Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a static method using the **@staticmethod decorator** or **staticmethod() function**
- Static methods are defined inside a class, and it is pretty similar to defining a regular function. To declare a static method, use this idiom:

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

Example: Create Static Method Using @staticmethod Decorator

- To make a method a static method, add @staticmethod decorator before the method definition
- The @staticmethod decorator is a built-in function decorator in Python to declare a method as a static method. It is an expression that gets evaluated after our function is defined
- In this example, we will create a static method gather_requirement() that accepts the project name and returns all requirements to complete under this project
- Static methods are a special case of methods
- Sometimes, you'll write code that belongs to a class, but that doesn't use the object itself at all. It is a utility method and doesn't need an object (self parameter) to complete its operation
- So we declare it as a static method. Also, we can call it from another method of a class

```
class Employee(object):

    def __init__(self, name, salary, project_name):
        self.name = name
        self.salary = salary
        self.project_name = project_name

    @staticmethod
    def gather_requirement(project_name):
        if project_name == 'ABC Project':
            requirement = ['task_1', 'task_2', 'task_3']
        else:
            requirement = ['task_1']
        return requirement

    # instance method
    def work(self):
        # call static method from instance method
        requirement = self.gather_requirement(self.project_name)
```

```
        for task in requirement:
            print('Completed', task)

    emp = Employee('Kelly', 12000, 'ABC Project')
    emp.work()
```

Output:

```
Completed task_1
Completed task_2
Completed task_3
```

C.3. Advantages of a Static Method:

- Here, the static method has the following advantages:

- **Consume Less memory**: Instance methods are object too, and creating them has a cost. Having a static method avoids that. Let's assume you have ten employee objects and if you create `gather_requirement()` as a instance method then Python have to create a ten copies of this method (separate for each object) which will consume more memory. On the other hand static method has only one copy per class

```
kelly = Employee('Kelly', 12000, 'ABC Project')
jessa = Employee('Jessa', 7000, 'XYZ Project')

# false
# because seperate copy of instance method is created for each object
print(kelly.work is jessa.work)

# True
# because only one copy is created
# kelly and jess objects share the same methods
print(kelly.gather_requirement is jessa.gather_requirement)

# True
print(kelly.gather_requirement is Employee.gather_requirement)
```

- **To Write Utility functions**: Static methods have limited use because they don't have access to the attributes of an object (instance variables) and class attributes (class variables). However, they can be helpful in utility such as conversion form one type to another. The parameters provided are enough to operate

- **Readabiltity**: Seeing the `@staticmethod` at the top of the method, we know that the method does not depend on the object's state or the class state

C.4. The staticmethod() function:

- Some code might use the old method of defining a static method, using `staticmethod()` as a function rather than a decorator
- You should only use `staticmethod()` function to define static method if you have to support older versions of Python (2.2 and 2.3). Otherwise, it is recommended to use the `@staticmethod` decorator

Syntax:

```
staticmethod(function)
```

- `function`: It is the name of the method you want to convert as a static method
- It returns the converted static method

Example:

```
class Employee:
    def sample(x):
        print('Inside static method', x)

# convert to static method
Employee.sample = staticmethod(Employee.sample)
# call static method
Employee.sample(10)
```

- The `staticmethod()` approach is helpful when you need a reference to a function from a class body and you want to avoid the automatic transformation to the instance method

- Call Static Method from Another Method
- Let's see how to call a static method from another static method of the same class. Here we will class a static method from a class method

```
class Test :
    @staticmethod
    def static_method_1():
        print('static method 1')

    @staticmethod
    def static_method_2() :
        Test.static_method_1()

    @classmethod
    def class_method_1(cls) :
        cls.static_method_2()

# call class method
Test.class_method_1()
```

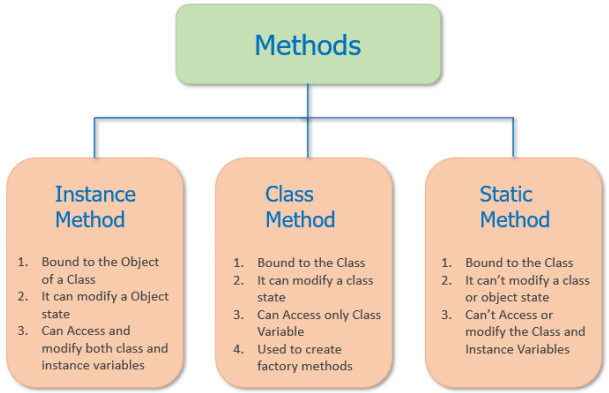
Output:

```
static method 1
```

- **Python Class Method vs. Static Method vs. Instance Method**:

Types of methods in python classes

- In Object-oriented programming, when we design a class, we use the following three methods:
  - Instance method performs a set of actions on the data/value provided by the instance variables. If we use instance variables inside a method, such methods are called instance methods
  - Class method is method that is called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method
  - Static method is a general utility method that performs a task in isolation. This method doesn't have access to the instance and class variable



class method vs static method vs instance method

Difference #1: Primary Use:

- Class method Used to access or modify the class state. It can modify the class state by changing the value of a class variable that would apply across all the class objects
- The instance method acts on an object's attributes. It can modify the object state by changing the value of instance variables
- Static methods have limited use because they don't have access to the attributes of an object (instance variables) and class attributes (class variables). However, they can be helpful in utility such as conversion form one type to another
- Class methods are used as a factory method. Factory methods are those methods that return a class object for different use cases. For example, you need to do some pre-processing on the provided data before creating an object

Difference #2: Method Definition:

▸ Let's learn how to define instance method, class method, and static method in a class. All three methods are defined in different ways.

- All three methods are defined inside a class, and it is pretty similar to defining a regular function
- Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a class method or static method
- Use the `@classmethod` decorator or the `classmethod()` function to define the class method
- Use the `@staticmethod` decorator or the `staticmethod()` function to define a static method

  Example:
  - Use `self` as the first parameter in the instance method when defining it. The `self` parameter refers to the current object
  - On the other hand, Use `cls` as the first parameter in the class method when defining it. The `cls` refers to the class
  - A static method doesn't take instance or class as a parameter because they don't have access to the instance variables and class variables

```
class Student:
    # class variables
    school_name = 'ABC School'

    # constructor
    def __init__(self, name, age):
        # instance variables
        self.name = name
        self.age = age

    # instance method
    def show(self):
        print(self.name, self.age, Student.school_name)

    @classmethod
    def change_School(cls, name):
        cls.school_name = name

    @staticmethod
    def find_notes(subject_name):
        return ['chapter 1', 'chapter 2', 'chapter 3']
```

As you can see in the example, in the instance

Difference #3: Method Call:

- Class methods and static methods can be called using ClassName or by using a class object
- The Instance method can be called only using the object of the class

Example:

```
# create object
jessa = Student('Jessa', 12)
# call instance method
jessa.show()

# call class method using the class
Student.change_School('XYZ School')
# call class method using the object
jessa.change_School('PQR School')

# call static method using the class
Student.find_notes('Math')
# call class method using the object
jessa.find_notes('Math')
```

Output:

```
Jessa 12 ABC School
School name changed to XYZ School
School name changed to PQR School
```

Difference #4: Attribute Access:

- Both class and object have attributes. Class attributes include class variables, and object attributes include instance variables
  - The instance method can access both class level and object attributes. Therefore, It can modify the object state
  - Class methods can only access class level attributes. Therefore, It can modify the class state
  - A static method doesn't have access to the class attribute and instance attributes. Therefore, it **cannot** modify the class or object state

Example:

```
class Student:
    # class variables
    school_name = 'ABC School'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def show(self):
        # access instance variables
        print('Student:', self.name, self.age)
        # access class variables
        print('School:', self.school_name)

    @classmethod
    def change_School(cls, name):
        # access class variable
        print('Previous School name:', cls.school_name)
        cls.school_name = name
        print('School name changed to', Student.school_name)

    @staticmethod
    def find_notes(subject_name):
        # can't access instance or class attributes
        return ['chapter 1', 'chapter 2', 'chapter 3']
```

```
# create object
jessa = Student('Jessa', 12)
# call instance method
jessa.show()

# call class method
Student.change_School('XYZ School')
```

Output:

```
Student: Jessa 12
School: ABC School
Previous School name: ABC School
School name changed to XYZ School
```

Difference #5: Class Bound and Instance Bound:

- An instance method is bound to the object, so we can access them using the object of the class
- Class methods and static methods are bound to the class. So we should access them using the class name

Example:

```
class Student:
    def __init__(self, roll_no): self.roll_no = roll_no

    # instance method
    def show(self):
        print('In Instance method')

    @classmethod
    def change_school(cls, name):
        print('In class method')

    @staticmethod
    def find_notes(subject_name):
        print('In Static method')

# create two objects
jessa = Student(12)

# instance method bound to object
print(jessa.show)

# class method bound to class
print(jessa.change_school)

# static method bound to class
print(jessa.find_notes)
```

Do you know:

- In Python, a separate copy of the instance methods will be created for every object
- Suppose you create five Student objects, then Python has to create five copies of the `show()` method (separate for each object)
- So it will consume more memory. On the other hand, the static method has only one copy per class

Example:

```
# create two objects
jessa = Student(12)
kelly = Student(25)

# False because two separate copies
print(jessa.show is kelly.show)

# True objects share same copies of static methods
```

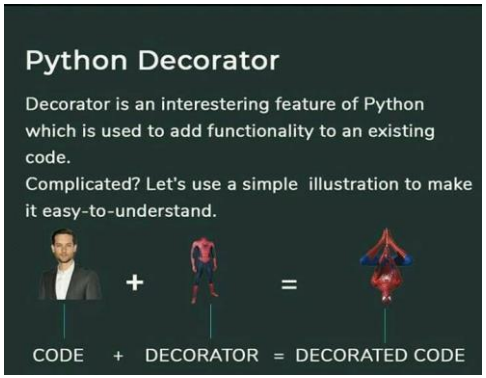
```
print(jessa.find_notes is kelly.find_notes)
```

```
Jessa 20 ABC School
Jessa 20 XYZ School
<bound method Student.change_School of <class '__main__.Student'>>
```

As you can see in the output, the change_School() method is bound to the class

Instance Attribute vs Class Attribute AND Instance Method vs class Method vs Static Method:

- Attributes are normal variables but just because they are in a class they are called instance variables, they belong to an object otherwise they are standalone variables
- Methods are actually normal functions but just because they are in a class they are called methods
- We have instance attributes that we can define in or out of the class AND class attributes that are defined inside the class before the constructor
- We can change the instance variables & class attributes through instance methods or from outside of the class
- We can combine attribute instances in the constructor to create something new like an email (made from first and last name)
- Once you create a method it will automatically be instance method BUT you can convert it into classmethod or staticmethod through decorators that will change the behavior of the function
- Good habit to access the classmethod is through calling the class name not the object (even if it is possible)
- Instance method takes 'self', classmethod takes 'cls' and static method takes what argument we want to give it to work with
- If you do not want to declare all parameters in constructor then you can use for ex. first:str="", number:int=0
- You can pass parameters in the constructor, instance method, classmethod and staticmethod
- format as a string the constructor by using __repr__ method for better readout otherwise it will show where in memory it was stored
- It is possible to print directly the class with its class attribute -> print(Pizza(["Mozarella","Ham"]))
- instance variable will be assigned to 'self' object
- override/add constructor parameters using classmethod from the caller
- a giveaway that a method should be a staticmethod is that if you don't access the instance or the class anywhere within the function ex. 'cls' or 'self' then it is staticmethod
- staticmethod does not have access to the object or the instance at all, its independent from everything around it
- _ is used to mark it as not being public for this API rather an internal implementation detail
- Use assert after constructor for debug -> assert price >=0,f"...."
- After constructor we can execute actions like append to a list or count how many objects created
- With 'self' in front of an instance method, it means that the attribute can be accessed from level class or instance level, so read from instance level or class level
- classmethod can only accessed from the class level
- with classmethod we will not have any instances on our hand to call this method from the instance because this method is designed for instantiating the object itself
- by removing 'self' and adding 'cls', the class reference will be passed as first argument
- you can use context manager with classmethod, ex.to read the items from csv file
- classmethod will do something that has a relationship with the class, but usually, those are used to manipulate different structures of data to instantiate objects, like with csv files
- classmethod will work with subclasses, extremely used, used for alternative constructors, are usually called 'from, make or create' something
- classmethod purpose is to make instance using different set of arguments and used to create factory methods (factory methods return class objects like constructor)
- classmethod should do something that has a relationship with the class but usually, those are used to manipulate different structures of data to instantiate objects, like we have done with csv
- staticmethod create utility functions, this should also do something that has a relationship with the class, but not something that must be unique per instance, uncommon to be used
- staticmethod sometimes you might be better off writing it standalone function rather than static method unless it is really linked only to that class
- staticmethod should do something that has a relationship with the class, but not something that must be unique per instance
- so staticmethod is just a utility, is related to the class but does not use instance or attributes for example if we want to check the expiry of the beans in a store, we will create a utility to do so
- You have instance level and class level, first the object will look for the instance variable in the instance level then if not found will look in the class level




```python
class MyClass:
"""Instance method vs classmethod vs staticmethod"""

    def method(self):
        # can modify object instance state
        # can modify class state
        return 'instance method called', self

    @classmethod
    # can't modify object instance state
    # can modify class state
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    # can't modify object instance state
    # can't modify class state
    def staticmethod():
        return 'static method called'

obj = MyClass()
print(obj.method())
obj = MyClass()
print(obj.classmethod())
obj = MyClass()
print(obj.staticmethod())

print(MyClass.classmethod())
print(MyClass.staticmethod())
```

```python
class Employee:
    """change class variable with classmethod,
    construct email from first and last name to be @gmail,
    create full name, use classmethod to increase the class
    attribute raise of salary"""

    number_of_employee = 0
    raise_salary = 1.05

    #constructor
    def __init__(self, first:str="", last:str="", pay:int=0):
        # instance variables
        self.first = first
        self.last = last
        self.pay = pay
        # combine instance attributes
        self.email = first+'.'+last+'@gmail.com'
```

```
        # counting the number of the employee
        Employee.number_of_employee +=1

    # instance method
    def full_name(self):
        """create full name """
        return '{} {}'.format(self.first, self.last)

    #instance method
    def apply_raise(self):
        """Increase instance variable pay by using raise salary class attribute"""
        self.pay = int(self.pay * self.raise_salary)

    # decorator -> acts as a converter
    @classmethod
    # classmethod will take the class method and replace it with the caller input amount
    def set_raise_amount(cls, amount):
        """classmethod to increase the class attribute raise initial value"""
        cls.raise_salary = amount

emp_1 = Employee('Corey', 'Schaffer',5000)
print(emp_1.full_name())
print(emp_1.email)
print(emp_1.pay) # 5000
emp_1.apply_raise()
print(emp_1.pay) #5250 -> money after increase of slary
Employee.set_raise_amount(50)
emp_1.apply_raise()
print(emp_1.pay)
print(Employee.number_of_employee) # 1
emp_2 = Employee()
print(Employee.number_of_employee) # 2
```



```
class Pizza:
    """Change Constructor using classmethod
        To create pizza type with its ingredients"""

    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        #format it as a string
        return f"Pizza({self.ingredients})"

    @classmethod
    def margherita(cls):
        return print(cls(['cheese','tomatoes']))

    @classmethod
    def prosciutto(cls):
        return print(cls(['cheese','tomatoes','ham','mushrooms']))

# print(Pizza(['cheese','tomatoes']))
# print(Pizza(['cheese','tomatoes','ham']))
# print(Pizza(['cheese','tomatoes','chorizo','mushrooms']))

Pizza.margherita()
Pizza.prosciutto()
```



```
class Employee:
    """ change class variable with classmethod,
    construct email from first and last name to be @gmail,
    create full name, use classmethod to increase the class
```

```
    attribute raise of salary, make a method to split the info
    if received with hyphen"""

    number_of_employee = 0
    raise_salary = 1.05

    #constructor
    def __init__(self, first:str="", last:str="", pay:int=0):
        # instance variables
        self.first = first
        self.last = last
        self.pay = pay
        # combine instance attributes
        self.email = first+'.'+last+'@gmail.com'

        # counting the number of the employee
        Employee.number_of_employee +=1

    # instance method
    def full_name(self):
        """create full name """
        return '{} {}'.format(self.first, self.last)

    #instance method
    def apply_raise(self):
        """Increase instance variable pay by using raise salary class attribute"""
        self.pay = int(self.pay * self.raise_salary)

    # decorator -> acts as a converter
    @classmethod
    # classmethod will take the class method and replace it with the caller input amount
    def set_raise_amount(cls, amount):
        """classmethod to increase the class attribute raise initial value"""
        cls.raise_salary = amount

    @classmethod
    def split_string(cls, employee_string):
        first, last, pay = employee_string.split('-')
        return cls(first,last,pay)

emp_1 = Employee('Corey', 'Schaffer',5000)
print(emp_1.full_name())
print(emp_1.email) #Corey.Schaffer@gmail.com
print(emp_1.pay) # 5000
emp_1.apply_raise()
print(emp_1.pay) #5250 -> money after increase of salary
Employee.set_raise_amount(50)
emp_1.apply_raise()
print(emp_1.pay)
print(Employee.number_of_employee) # 1
emp_2 = Employee()
print(Employee.number_of_employee) # 2
employee_string_1 = "John-Doe-6000"
#first , last, pay = emp_str_1.split('-')#no need now as we have the def from_string
#new_emp_1 = Employee(first,last,pay)#no need now as we have the def from_string
new_employee = Employee.split_string(employee_string_1) # this will do same as above, so no need to parse the strings anymore
print(new_employee.pay) #6000
```

```
class Pizza:
        """Instance Attribute vs classmethod
            calculate total price using instance attributes,
            apply discount using instance method with
            instance attribute and class attribute,
            change the pizza price of the class using classmethod,
            add quantity to the class using classmethod,
```

```
            change pizza type using classmethod """

    pizza_price = 20 # class variables
    quantity = [] # we will add our instances here
    discount = 0.8

    def __init__(self, type_of_pizza):# constructor
        self.type_of_pizza = type_of_pizza #instance variable, # assign to self object
        self.price = 3
        self.store_price = 3

    def __repr__(self):
        return f"{self.type_of_pizza}"# so write it like you are writing when you instantiated the objects

    def calculate_total_price(self): # instance methods operations using instance attributes
        return print(self.price * self.store_price)

    def apply_discount(self): # instance method to change the value of instance attribute by using class attribute
        self.price = self.price * self.discount
        return print(self.price)

    @classmethod # classmethod that will change my class attribute to increase the price
    def change_price(cls):
        cls.pizza_price = 50
        return print(cls.pizza_price)

    @classmethod # adding elements to class attribute
    def add_quantity(cls, add_quantity):
        cls.quantity.append(add_quantity)

    @classmethod #change constructor using classmethod from caller overriding the constructor
    def change_pizza_type(cls,type_of_pizza):
        return print(cls(type_of_pizza))

    @classmethod #change constructor using classmethod from classmethod overriding the constructor
    def change_pizza_type(cls):
        return print(cls('Margarita'))


pizza_1 = Pizza("Texas") # create an instance
pizza_1.calculate_total_price() # instance methods operations using instance attributes

pizza_1.apply_discount()# instance method to change the value of instance attribute by using class attribute

print(Pizza.pizza_price) # class attribute value
Pizza.change_price() # apply class method to change class attribute
print(Pizza.pizza_price) # class attribute after using classmethod to change price

Pizza.add_quantity(10) # use class method to add to class attribute
print(Pizza.quantity) # print new class attribute list

print(Pizza("Prosciutto Funghi")) # printing constructor with the type of argument we initially have

#Pizza.change_pizza_type('Quattro Formaggi')#change constructor using classmethod from classmethod overriding the constructor from caller
Pizza.change_pizza_type() #change constructor using classmethod from classmethod overriding the constructor
```

```
import math

class Pizza:
    """Use staticmethod to a create a utility to be used
            To calculate the radius that will be used in an
            Instance method"""

    def __init__(self, radius, ingredients):
        self.ingredients = ingredients
```

```
        self.radius = radius

    def __repr__(self):
        return f'Pizza({self.ingredients})'

    def area(self):
        return self._circle_area(self.radius)

    @staticmethod # does not have access to the object or the instance at all, its independent from everything around it
    def _circle_area(r):
        return r ** 2 * math.pi

print(Pizza(4.5,['cheese']).area())
print(Pizza._circle_area(12)) #_ is used to mark it as not being public for this api rather an internal implementation detail
```



```
class LidI:
    """creating a utility staticmethod that the instance method will use
    along with the class attribute to determine if the product is expired"""

    def __init__(self,expiry):
        self.expiry = expiry

    def __repr__(self):
        return f"{self.expiry}"

    def check_expiry(self):
        return LidI.expiry_date(self.expiry) # notice we used self so the object will access the utility

    @staticmethod
    def expiry_date(date_of_expiration):
        if date_of_expiration == 2022:
            print("the products is ok")
        else:
            print("the products is expired")

beans = LidI(2020)
beans.check_expiry() #the products is expired
```



```
class Employee:
    """instance method using staticmethod to inject
       into the classmethod parameters"""

    def __init__(self, name, salary,project_name):
        self.name = name
        self.salary = salary
        self.project_name = project_name

    @staticmethod
    def gather_requirement(project_name):
        """utility method to check which task
          is completed based on the project """
        if project_name == 'ABC Project':
            requirement = 'task 1'
        else:
            requirement = 'task 2'
        return requirement

    # we use the staticmethod and inject the classmethod as parameter
    def work(self):
        """instance method that will use staticmethod to check requirement task
        using the statismethod"""
        requirement = Employee.gather_requirement(self.project_name)
        print("Completed", requirement)
```

```
emp = Employee("lulu",12000,'ABC Project')
emp.work()
```



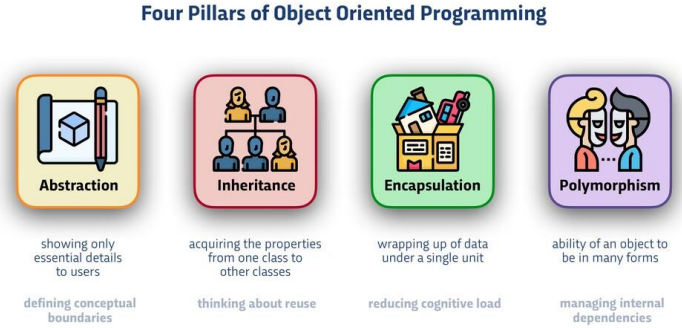
```
class Employee:
    # Docstring
    """create email instance from instance variable,
    count how many employees we have, use intance method
    to create full name"""

    # class attributes
    number_of_employee = 0
    raise_amount = 1.04

    # constructor
    def __init__(self, first, last, pay):
        # instance attributes
        self.first = first
        self.last = last
        self.pay = pay
        #create instance variable email using first and last name instances
        self.email = first+"."+last

        # count how many employee
        Employee.number_of_employee += 1

    # instance method
    def full_name(self):
        """join first and last name"""
        return f"{self.first} {self.last}"

    # instance method
    def apply_raise(self):
        """raise the salary pay by using the instance variable
        and class variable"""
        self.pay = self.pay * self.raise_amount

    # classmethod
    @classmethod
    def change_raise_amount(cls, new_raise_amount):
        """increase the raise of the main raise from the class attribute
        by using a classmethod to change the class attribut and taking
        the argument from the caller"""
        cls.raise_amount = new_raise_amount

    # classmethod
    @classmethod
    def get_details_from_string(cls, employee_string):
        """alternative constructor if the employee
        will use hyphen to enter the name, we will have to
        seperate the first,last name and the pay"""
        first, last, pay = employee_string.split("-")
        return cls(first, last, pay)

    # staticmethod
    # a give away that a method should be a staticmethod is if you don't access the instance or the class anywhere within the function ex. cls or self
    # static methods don't take the instance or the class as the first argument so we can pass in the arguments we want to work with
    @staticmethod
    def check_if_working_day(day):
        """ we are going to check if the day will falls on a weekday,
        in python dates have the weekdays methods where monday is 0
        and sunday 6 and other days in between,
        so is it a working day? if no then False if it is a working day
        then True"""
        if day.weekday() == 5 or day.weekday()== 6:
```

```
        # so if the day will be a Saturday or Sunday
        return False
    return True


employee_1 = Employee("Corey","Scafer",5000)
print(Employee.number_of_employee) # 1 Employee
print(employee_1.full_name()) # join first and last name
print(employee_1.pay) # 5000
employee_1.apply_raise() # increase the salary
print(employee_1.pay) # 5200 -> salary has increased
Employee.change_raise_amount(20) # change the main raise of salaries tp 20
print(employee_1.pay) # 5200
employee_1.apply_raise() # apply raise after the change of main salary increase
print(employee_1.pay) # 104000
employee_1_string = Employee.get_details_from_string("Michael-Jackson-6000")
print(employee_1_string.first) # Michael

import datetime

possible_date_to_work = datetime.date(2022, 8, 27)
print(Employee.check_if_working_day(possible_date_to_work))# False because it is Saturday
#help(Employee)
```

```
import csv

class Item:

    pay_rate = 0.8 # the pay rate after .20 discount
    all = []

    def __init__(self, name:str, price:float, quantity):
        assert quantity >= 0, f"quantity {quantity} is not greater or equal to 0"
        self.name = name
        self.price = price
        self.quantity = quantity

        # Actions to execute
        Item.all.append(self)

    def calculate_total_price(self):
        return self.price * self.quantity

    def apply_discount(self):
        self.price = self.price * self.pay_rate

    @classmethod
    def instantiate_from_csv(cls):
        with open('tems2.csv','r') as f:
            reader = csv.DictReader(f)
            items = list(reader)

        for item in items:
            Item(
                name=item.get('name'),
                price=float(item.get('price')),
                quantity=int(item.get('quantity')),
            )

    @staticmethod
    def is_integer(num):
        # we will count out the floats that are point zero
        if isinstance(num, float):
            #count out the floats that are zero
            return num.is_integer()
```

```
    elif isinstance(num, int):
        return True
    else:
        return False

def __repr__(self):
    return f"Item'{self.name}', {self.price}, {self.quantity}"

print(Item.is_integer(7.0))
```

C.6.2.3. Encapsulation, Abstraction, Inheritance, Polymorphism:



➢ **The Four Pillars of Object Oriented Programming:**

- The pillars are:
  - **I. Encapsulation (Data hide)**
  - **II. Data Abstraction (Data hide)**
  - **III. Inheritance (Reusability)**
  - **IV. Polymorphism (Object to take many forms)**
- These pillars are fundamental to OOP and how you are already incorporating them into your code
- These pillars keep our work organized, modular and maintainable. They stop us from repeating ourselves, keep our data safe and most importantly make it simpler for our colleagues and our users to interact with. They hold up the object oriented pantheon in all their doric glory so you don't forget…

• In order to understand the pillars of OOP let us first define what OOP is:

- Object Oriented Programming is a language model that is organized around **objects** rather than **actions** and **data** rather than **logic**
  - To be clear, that does not mean we do not *use* actions and logic in our code(or at least we should), but that rather than coding out a set of linear instructions in which we pass function to function or input to output we instead build objects. We model things that have attributes. They hold specific types and quantities of data and have methods they can perform. This allows our objects to interact with each other in order to read, write, interpret, calculate, manipulate or otherwise interact with that data

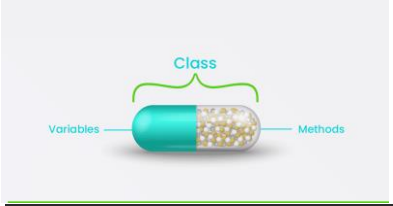
So how and why do we do all of this? That is where the four pillars come into play. Here they are:

||

A.P.I.E. Which is easier to remember. Because A Pie!

I. Encapsulation:

- **A capsule (which we consume when we are ill) hide/bind some powder from in itself, means that capsule encapsulate the powder contained it**
  = the system of wrapping data and functions together



There are 2 meanings for Encapsulation

1. The first is that we group things in a way that each group represents the essential features of something

- Example: a culture document encapsulates a company's beliefs values and practices
  customer class represents all the data we consider to be essential
  for representing a customer in our software system

```
class Customer:
  id: str
  name: str
  email_address: str
  address_line_1: str
  address_line_1: str
  postal_code : str
```

city: str
country: str
• So this list will be complete eventually, where you will want to encapsulate everything that you want a customer to represent

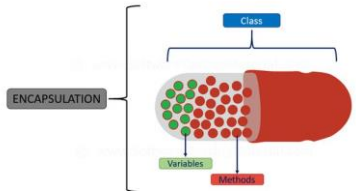
2. Second meaning of encapsulation is that it defines boundaries round things

- Example: • Hans solo was encapsulated in carbonite by Jabba the Hutt
  • or encapsulated in prison by the government
  - So encapsulation in that sense is about restricting access in some way
- In code you can achieve this by making instance variables and methods private or protected
  - Example: your software is using a payment or stripe, you want to be able to do payment processing at various places in your code but you don't want that code to be dependent on stripe specific things what do you do? you hide information, you create a stripe payment processing module that contains functions for starting a payment processing refunds , calculating tax percentages etc inside the module you have for the implementation details such as which stripe specific API calls you should make authenticating to the stripe surface extracting the data and transforming it into a format that your application can use so on
  - All the other parts of your code use the stripe payment processor module that you've built and you don't need to know anything about the implementation details that information is hidden, in that sense the strike payment processor module acts as facade which is one of the design patterns in the gang of four book

- Other example: you go to the bank and give them 5000, after 1 year they repay you with 10000 but you don't know how much profit they made maybe 5 times more , you never know it so you never know what object of the class has done with the variables of the class

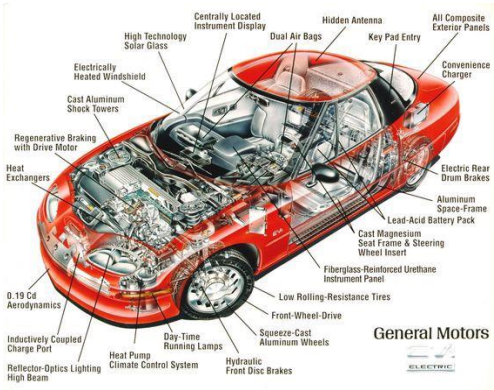
So, rule of encapsulation class:
Declare private instances variable
Public setter and getter methods
ex. you can return money*2

- Data is kept hidden which cannot be accessed directly outside the object although it is available in the same program
- The object maintains privacy of the data members, However, the changes that take in method don't affect other objects
- Encapsulation = hiding data and complexity
- All what we need to know when accessing a protected class is the name of the class, the properties, and methods available and any data to be supplied when we they need to be called
- All what our user will access our code will be actually the interface **->** the implementation for the person to access that class will remain a mystery



- Think of a car:
  • **Most likely you thought of something along the lines of this:**

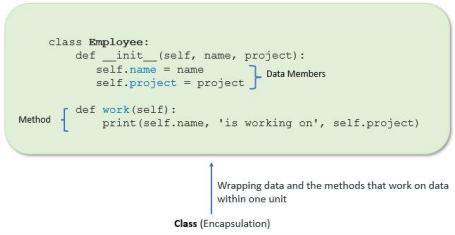
• **Probably <u>not</u> this:**

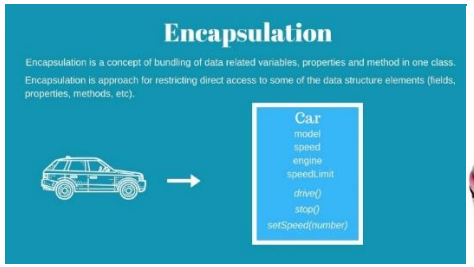


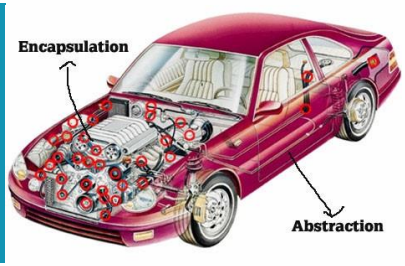
- **We keep all the inner workings of the car together encapsulated under the hood**
- **The important stuff is stored there to keep it safe from the average user(non mechanics) which ensures the integrity of the system as it was designed**
- If you've ever incurred hefty mechanic fees after messing with something inside your engine then you understand the value and need for encapsulation, we don't want this to happen to our code either
- Take this code block for example:
  - We have defined a simple **Person class**. We encapsulate the data(first name, last name, age) and the functions needed for that data(getter methods for names and a format method for date) inside the class
  - We have packaged all relevant information together and only the methods within that package can directly manipulate that information
  - This concept will lead us to further restrict access within our classes to not only maintain the integrity and security of our software but also to facilitate abstraction
- Abstraction and encapsulation are tightly related
- Stepping back into the car analogy, when we interact with a car we only need to know that the gas is on the right, the break is on the left, put it in drive and the steering wheel tells the car where to go **<-<u>This is abstraction</u>**
  -> We are only showed the simplest possible interface and the non-essential information to that interface has been hidden.

I.A. What is Encapsulation in Python?

- Encapsulation in Python describes the concept of **bundling data and methods within a single unit, it is the mechanism that binds code and data that it manipulates**
- Encapsulation is the protective shield that prevents the data from being accessed by the code outside this shield, it is like the powder inside a capsule or a vending machine
  - So, for example, when you create a class, it means you are implementing encapsulation
  - A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit







- **Example**:
  - In this example, we create an Employee class by defining employee attributes such as name and salary as an instance variable and implementing behavior using work() and show() instance methods

```
class Employee:
    # constructor
    def __init__(self, name, salary, project):
        # data members
        self.name = name
        self.salary = salary
        self.project = project

    # method
    # to display employee's details
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

    # method
```

```
    def work(self):
        print(self.name, 'is working on', self.project)

# creating object of a class
emp = Employee('Jessa', 8000, 'NLP')

# calling public method of the class
emp.show()
emp.work()
```

Output:

```
Name:  Jessa Salary: 8000
Jessa is working on NLP
```

- **Using encapsulation, we can hide an object's internal representation from the outside. This is called information hiding**
- **Also, encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class**
  - Encapsulation is a way to can restrict access to methods and variables from outside of class.
  - Whenever we are working with the class and dealing with sensitive data, providing access to all variables used within the class is not a good choice.
  - For example, Suppose you have an attribute that is not visible from the outside of an object and bundle it with methods that provide read or write access

  -> In that case, you can hide specific information and control access to the object's internal state

  -> Encapsulation offers a way for us to access the required variable without providing the program full-fledged access to all variables of a class

  -> This mechanism is used to protect the data of an object from other objects

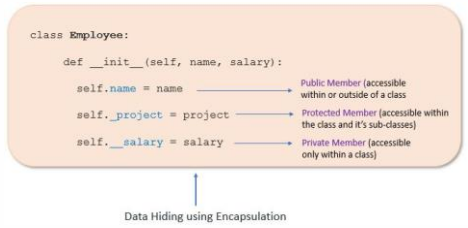
I.B. Access Modifiers in Python:

- **Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected**
- But In Python, we don't have direct access modifiers like public, private, and protected. We can achieve this by using single **underscore** and **double underscores**
- Access modifiers limit access to the variables and methods of a class
- Python provides three types of access modifiers private, public, and protected:

  i) Public Member: Accessible anywhere from outside a class

  ii) Private Member: Accessible within the class

  iii) Protected Member: Accessible within the class and its sub-classes



i) Public Member

- Public data members are accessible within and outside of a class. All member variables of the class are by default public

Example:

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data members
        self.name = name
        self.salary = salary

    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing public data members
```

```
print("Name: ", emp.name, 'Salary:', emp.salary)

# calling public method of the class
emp.show()
```

Output

```
Name:  Jessa Salary: 10000
Name:  Jessa Salary: 10000
```

ii) Private Member

- We can protect variables in the class by marking them private. To define a private variable add two underscores as a prefix at the start of a variable name
- Private members are accessible only within the class, and we can't access them directly from the class objects

Example:

```
class Employee:
   # constructor
   def __init__(self, name, salary):
      # public data member
      self.name = name
      # private member, with 2 underlines
      self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing private data members
print('Salary:', emp.__salary)
```

Output

```
AttributeError: 'Employee' object has no attribute '__salary'
```

- **In the above example, the salary is a private variable.** As you know, we can't access the **private variable** from the outside of that class
- We can access private members from outside of a class using the following two approaches:
  - **a) Create public method to access private members**
  - **b) Use name mangling**

➢ **Let's see each one by one:**

a) Public method to access private members:

Example: Access Private member outside of a class using an instance method

```
class Employee:
   # constructor
   def __init__(self, name, salary):
      # public data member
      self.name = name
      # private member
      self.__salary = salary

   # public instance methods
   def show(self):
      # private members are accessible from a class
      print("Name: ", self.name, 'Salary:', self.__salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# calling public method of the class
emp.show()
```

Output:

```
Name: Jessa Salary: 10000
```

b) Name Mangling to access private members:

- We can directly access private and protected variables from outside of a class through name mangling
- The name mangling is created on an identifier by adding two leading underscores and one trailing underscore, like this `_classname__dataMember`, where `classname` is the current class, and data member is the private variable name

Example: Access private member

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

print('Name:', emp.name)
# direct access to private member using name mangling
print('Salary:', emp._Employee__salary)
```

Output

```
Name: Jessa
Salary: 10000
```

iii) Protected Member:

- Protected members are accessible within the class and also available to its sub-classes. To define a protected member, prefix the member name with a single underscore `_`
- Protected data members are used when you implement inheritance and want to allow data members access to only child classes

Example: Protected member in inheritance:

```
# base class
class Company:
    def __init__(self):
        # Protected member
        self._project = "NLP"

# child class
class Employee(Company):
    def __init__(self, name):
        self.name = name
        Company.__init__(self)

    def show(self):
        print("Employee name :", self.name)
        # Accessing protected member in child class
        print("Working on project :", self._project)

c = Employee("Jessa")
c.show()

# Direct access protected data member
print('Project:', c._project)
```

Output

```
Employee name : Jessa
Working on project : NLP
```

```
Project: NLP
```

I.C. Getters and Setters in Python:

- To implement proper encapsulation in Python, we need to use setters and getters
- The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation
- Use the getter method to access data members and the setter methods to modify the data members
- In Python, private variables are not hidden fields like in other programming languages. The getters and setters methods are often used when:
  - When we want to avoid direct access to private variables
  - To add validation logic for setting a value

Example:

```
class Student:
    def __init__(self, name, age):
        # private member
        self.name = name
        self.__age = age

    # getter method
    def get_age(self):
        return self.__age

    # setter method
    def set_age(self, age):
        self.__age = age

stud = Student('Jessa', 14)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())

# changing age using setter
stud.set_age(16)

# retrieving age using getter
```

Output

```
Name: Jessa 14
Name: Jessa 16
```

- Let's take another example that shows how to use encapsulation to implement information hiding and apply additional validation before changing the values of your object attributes (data member)

Example: Information Hiding and conditional logic for setting an object attributes

```
class Student:
    def __init__(self, name, roll_no, age):
        # private member
        self.name = name
        # private members to restrict access
        # avoid direct data modification
        self.__roll_no = roll_no
        self.__age = age

    def show(self):
        print('Student Details:', self.name, self.__roll_no)

    # getter methods
    def get_roll_no(self):
        return self.__roll_no

    # setter method to modify data member
    # condition to allow data modification with rules
```

```
    def set_roll_no(self, number):
        if number > 50:
            print('Invalid roll no. Please set correct roll number')
        else:
            self.__roll_no = number

jessa = Student('Jessa', 10, 15)

# before Modify
jessa.show()
# changing roll number using setter
jessa.set_roll_no(120)


jessa.set_roll_no(25)
jessa.show()
```

Output:

```
Student Details: Jessa 10
Invalid roll no. Please set correct roll number

Student Details: Jessa 25
```

I.D. Advantages of Encapsulation:

- **Security**: The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification
- **Data Hiding**: The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method
  -> To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them
- **Simplicity**: It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other
- **Aesthetics**: Bundling data and methods within a class makes code more readable and maintainable

- **Encapsulation is wrapping of <u>data and methods</u> into a single unit which is class**
- **Main purpose of Encapsulation is restricted access to the data and methods of the class <u>BUT</u> you can create methods to access them outside the class (getters and setters) + use property decorators to remove () from caller**
- **from public to private you use prefix '__' <-2 underline**

```
class Hello:
    def __init__(self,name):
        self.a = 10
        self._b = 20
        self.__c = 30

hello = Hello('name')
print(hello.a) # will work
print(hello._b) # will work, partially protected, just a convention
print(hello.__c)#'Hello' object has no attribute '__c' # private
```



```
class Car:
    """To encapsulate our code we create functions"""
    def __init__(self, speed, color):
        self.__speed = speed
        self.__color = color

    # setter
    def set_speed(self,value):
        self.__speed = value

    #getter
    def get_speed(self):
        return self.__speed

ford = Car(200,'red')
honda = Car(250,'blue')
audi = Car(300,'black')

ford.set_speed(300)

ford.__speed = 400

ford.set_speed(400)

print(ford.get_speed())
print(ford.color)#'Car' object has no attribute 'color'
```



```
class Person:
    # we can write properties to read/write though those values, through getters and setters
    # by using @property and setter we can use same name for Def
    # Usually for encapsulation def we use capitalized letters
    def __init__(self, name, age, gender):
        self.__name = name
        self.__age = age
        self.__gender = gender

    @property
```

```
    def Name(self):
        """return the private data"""
        return self.__name

    @Name.setter
    def Name(self,value):
        """setting new value for our value
        but also setting conditions if we want"""
        if value =="Bob":
            self.__name = "Default"
        else:
            self.__name = value

p1 = Person("Mike",20,'m')
#print(p1.name)# 'Person' object has no attribute 'name'
print(p1.Name) # access will be done through getter
p1.Name = "Bob" # access will be done through setter not directly to the variable
print(p1.Name)
```


▸ If you use property decorator no need to put () when calling def get_name_of_product

```
class Lidl:

    def __init__(self, name_of_product, barcode):
        self.__name_of_product = name_of_product
        self.barcode = barcode

    @property
    def get_name_of_product(self):
        if self.__name_of_product == "Beans":
            return "not given "
        else:
            self.__name_of_product = "Something else"
            return self.__name_of_product

from test12 import Lidl

product_1 = Lidl("Beans", 895745648)
print(product_1.get_name_of_product)
```

```
class myClass:
    __a = 10 # private variable, so the scope is only within the class

    def disp(self):
        print(self.__a) # we used self because it is a class variable

obj = myClass()
obj.disp()

# print(myClass.__a) # error, we cannot access because it is a private variable # type object 'myClass' has no attribute '__a'
#################################
# private methods can be access ony within the method
class myClass:
    def __disp1(self): # private method
        print("This is display1 method")
    def disp2(self):
        print("This is display2 method")
        self.__disp1()

obj=myClass()
obj.disp2()
```

```python
# We can access private variables outside of class indirectly using methods
class myClass:
   __empid=101

   def getempid(self,eid):
      self.__empid = eid
   def dispempid(self):
      print(self.__empid)

obj=myClass()
obj.dispempid() # 101
obj.getempid(102)
obj.dispempid() #102
```

➢ **Encapsulation vs Abstraction**:

Abstraction:

1. **Process of picking the essence of an object you really need**
2. **In other words, pick the properties you need from the object Example:**
   a. TV - Sound, Visuals, Power Input, Channels Input
   b. Mobile - Button/Touch screen, power button, volume button, sim port
   c. Car - Steering, Break, Clutch, Accelerator, Key Hole
   d. Human - Voice, Body, Eye Sight, Hearing, Emotions

Encapsulation:

1. **Process of hiding the details of an object you don't need**
2. **In other words, hide the properties and operations you don't need from the object but are required for the object to work properly Example:**
   a. TV - Internal and connections of Speaker, Display, Power distribution b/w components, Channel mechanism
   b. Mobile - How the input is parsed and processed, How pressing a button on/off or changes volumes, how sim will connect to service providers
   c. Car - How turning steering turns the car, How break slow or stops the car, How clutch works, How accelerator increases speed, How key hole switch on/of the car
   d. Human - How voice is produced, What's inside the body, How eye sight works, How hearing works, How emotions generate and effect us

• ABSTRACT everything you need and ENCAPSULATE everything you don't need •

- **Abstraction :** Abstraction is the act of representing essential information without including background details and explanations
- **Encapsulation :** Encapsulation is the act of wrapping up of attributes(represented by data members) and operations (represented by functions) under one single unit (represented by class).
  - Taking **Real world example:**
    - Suppose you go to an *automatic cola vending machine* and request for a *cola*. The machine processes your request and gives the cola

- Here **automatic cola vending machine is a class.** It contains both **data i.e. Cola** can and **operations i.e. service mechanism** and they are wrapped/integrated under a single unit Cola Vending Machine. This is called **Encapsulation**
- **You need not know HOW the machine is working. This is called Abstraction**
- You can interact with cola can only through service mechanism. You cannot access the details about internal data like how much cans it contains, mechanism etc. This is **Data Hiding**
- You cannot pick the can directly. You request for cola through proper instructions and request mechanism (i.e. by paying amount and filling request) and get that cola only through specified channel. This is **message passing**
  The working and data is hidden from you. This is possible because that Vending machine is made (or Encapsulated or integrated) so. Thus, we can say **Encapsulation is a way to implement Abstraction**

- **Abstraction:** Suppose you are going to an ATM to withdraw money. You simply insert your card and click some buttons and get the money,
  You don't know what is happening internally on press of these buttons. Basically, you are hiding unnecessary information from user. So, abstraction is a method to hide internal functionalities from user
- **Encapsulation:** Suppose there is a tree. Now a tree can have its components like root, stem, branches, leaves, flowers and fruits,
  It has some functionalities like Photosynthesis. But in a single unit we call it a tree. In same way encapsulation is a characteristic to bind data members and functions in single unit

Parameter	Abstraction	Encapsulation
Use for	Abstraction solves the problem and issues that arise at the design stage.	Encapsulation solves the problem and issue that arise at the implementation stage.
Focus	Abstraction allows you to focus on what the object does instead of how it does it	Encapsulation enables you to hide the code and data into a single unit to secure the data from the outside world.
Implementation	You can use abstraction using Interface and Abstract Class.	You can implement encapsulation using Access Modifiers (Public, Protected & Private.)
Focuses	Focus mainly on what should be done.	Focus primarily on how it should be done.
Application	During design level.	During the Implementation level.

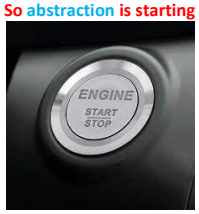
- Abstraction, encapsulation, etc are related terms in computer science. It is applicable to all kinds of programming, and not just Python or Object orientation
  - **Abstraction: is about ignoring non-essential features of something, to bring out the essential characteristics in relation to a focus point**
    - Let's take a simple example. You can talk about withdrawing money from a bank. This is an abstract concept. You could say that withdrawing $200 from a bank account will reduce your account by $200
    - Here we ignore **HOW** the withdrawal happened, and **focus on the essential aspect** of the **effect of the withdrawal** on your account
    - So we ignore whether you withdraw by cash, via ATM, via online payment, via cheque etc. **Abstraction is always based on a focus point; everything outside that focus is ignored**
    - For e.g., if we ignore environment, we can consider Whales and Human beings in a single abstraction (mammal). Here we ignore where the whale and human lives
  - **Encapsulation**: is a different concept, that allows you to implement abstraction. Note that, you can create abstractions without encapsulations, and you can encapsulate without necessarily creating an abstraction (or more correctly, the abstraction would be worthless)
    - **Encapsulation is about combining related things into a single whole.** Note that the whole can have parts; it is just that the whole can be considered as a single entity, or perhaps aggregate
    - Encapsulation brings with it many benefits, most importantly, allowing to easily reason about and manipulate numerous pieces of information as a whole. The human mind is limited after all
    - The simplest form of encapsulation is the file. You can divide your program into smaller pieces using different files. For e.g., for a game, we can divide it into:
      - graphics.py
      - game_logic.py
      - game_ai.py
      - game_networking.py
      - game_ui.py
        - Higher and higher forms of encapsulation were developed, for e.g., procedures, functions, modules, and finally objects
          - **Note how certain forms of encapsulation also leads to abstraction: when focusing on interface, rather than implementation**
            - You can **ignore how a function is implemented** and **consider only its signature** and calling semantics
              - Aggregates are also encapsulations of their constituents. We call them data structures
                - Data Structures focus on how groups of objects are structured and the behaviors surrounding their use
  - In OOP, **a class encapsulates both behavior and state**, in terms of member functions and member data (or attributes). **The class itself is an abstraction for its role in an object composition**

- **Abstraction: is providing an interface and hiding the unnecessary details**
  - Real world example - Car Engine , You **just move your key to start the car**. **All other details to start other components is taken care by the engine itself.** For you the **entire process has been abstracted as "Starting the Car"**
  - OOPs example - Class provide few methods for you to communicate. Private methods and variables are for internal processing which an outsider is least bothered. For you the entire process has been abstracted as public method process
- **Encapsulation:** is binding the body and actions together and protecting it from outside world
  - Real World Example - Again Car Engine, It bind together the different components like radiator , Batterry , starter and their operation into one component called engine. **It doesn't allow you to ignite**

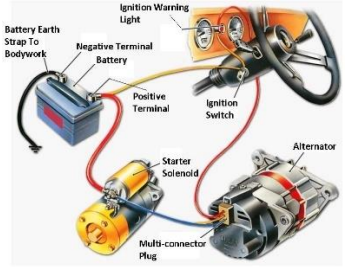
components individually

- OOPs - Class keeps only few methods as public which acts as interface whereas all other methods and variables are kept private to protect them from outside access

▸ **So abstraction is starting the car which will start the engine:**



▸ But **encapsulation** is the fact that the engine components and its operations will come under the engine that will do it for us without allowing us to control those components individually



Ignition System

▸ **Encapsulation:** is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as **private variable**

- Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section". Using encapsulation also hides the data. In this example, the data of the sections like sales, finance, or accounts are hidden from any other section

▸ Data Abstraction and Data Encapsulation both of these features are considered as the pillars of object-oriented programming language, but they have different functionality and implementation mechanism
Abstraction is a basic OOP concept which focuses on just the relevant data of an object and hides all the irrelevant details which may or may not be for generic or specialized behavior. It hides the background details and emphasizes on the essential points to reduce complexity and increase efficiency

▸ **Encapsulation** is yet another OOP concept which binds data and functions into a single component while restricting access to some components. It's one of the main fundamental concepts of OOP which wraps data and information under a single unit. In technical terms, encapsulation means hiding attributes to shield variables from outside access so that change in one part of an application won't affect the other parts

Abstraction	Encapsulation
It deals with only the relevant details by hiding the irrelevant ones to reduce complexity thereby increasing efficiency.	It binds the data and information together into a single entity to protect the data from external sources.
It refers to the idea of hiding data which is not required for presentation purposes.	It hides the data and code in order to restrict unwanted access.
It focuses on what rather than how.	It hides the internal mechanics of how it does something.
It hides the unnecessary details on the design level.	It also hides the details but on the implementation level.
Information and data is separated from the relevant data.	Information is hidden inside a capsule for close access.
It deals with ideas rather than events.	The idea is to protect the data from outside world.
It's implemented using abstract class and interface.	It's implemented using protected, private, and package-private access modifiers.

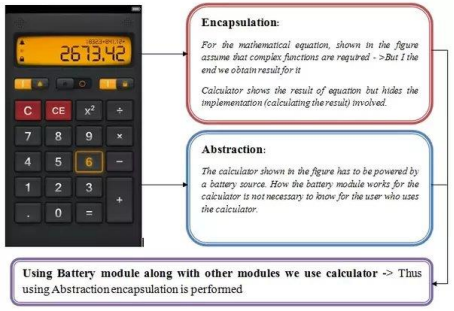
▸ Python implements weak encapsulation- that is it is encapsulation by convention, rather than being enforced by the language

- **Example:**

```
1.    class Colour:
2.        def __init__( self, r,g,b):
3.            self._r = r
4.            self._g = g
5.            self._b = b
6.
7.        def __abs__( self ):
8.            return int((self._r + self._g + self._b)/3)
9.
10.       @property
11.       def r(self):
12.           return self._r
13.
14.       @r.setter
15.       def r(self, value):
16.           if 0 > value or value > 255:
17.               raise ValueError('r component is out of range - must be between 0 and 255 inclusive')
18.           self._r = value
```

- Here the _r attribute is private by convention - other developers will know that it is intended to be private.
  - It is kept within a sensible range by the r.setter method which raises an error if the value is outside the 0 to 255 inclusive range
- There are also methods which work on that private attribute (such as the __abs__) method
  - In Python there is nothing to stop a developer accessing _r attribute directly, but that means that their code is now dependent on the internal design of the class and therefore not a good idea

- **Developer A**, who is **inherently utilizing** the concept of **abstraction** will use a module/library function/widget, concerned only with ***what* it does** (and *what* it will be used for) **but not *how* it does it**
  - **The interface of that module/library function/widget (the 'levers' the Developer A is allowed to pull/push) is the personification of that abstraction**
- **Developer B**, who is **seeking to create** such a module/function/widget will utilize the concept of **encapsulation** to ensure Developer A (and any other developer who uses the widget) can take advantage of the resulting **abstraction**. Developer B is most certainly concerned with *how* the widget does what it does



- Abstraction **- I care about *what* something does, but not *how* it does it**
- Encapsulation **- I care about *how* something does what it does such that others only need to care about *what* it does**

(As a loose generalization, to abstract something, you must encapsulate something else. And by encapsulating something, you have created an abstraction.)

- Encapsulation: • Is the packing of *"data"* and *"functions operating on that data"* into a single component and restricting the access to some of the object's components
  • Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition
- Abstraction: is a mechanism which represent the essential features without including implementation details

Encapsulation: -- **Information hiding**
Abstraction: -- **Implementation hiding**

- Encapsulation: puts some things in a box and gives you a peephole; this keeps you from mucking with the gears
- Abstraction: flat-out ignores the details that don't matter, like whether the things have gears, ratchets, flywheels, or nuclear cores; they just "go"

Examples of encapsulation:

- underpants
- toolbox
- wallet

- handbag
- capsule
- frozen carbonite
- a box, with or without a button on it
- a burrito (technically, the tortilla around the burrito)
examples of abstraction:
- "groups of things" is an abstraction (which we call aggregation)
- "things that contains other things" is an abstraction (which we call composition)
- "container" is another kind of "things that contain other things" abstraction; note that all of the encapsulation examples are kinds of containers, but not all containers exhibit/provide encapsulation. A basket, for example, is a container that does not encapsulate its contents

A. Short Answer:

- **Encapsulation** - Hiding and/or restricting access to certain parts of a system, while exposing the necessary interfaces
- **Abstraction** - Considering something with certain characteristics removed, apart from concrete realities, specific objects, or actual instances, thereby reducing complexity
  - The main **similarity** is that these techniques *aim to improve comprehension and utility*
  - The main **difference** is that *abstraction is a means of representing things more simply (often to make the representation more widely applicable), whereas encapsulation is a method of changing the way other things interact with something*

B. Long Answer:

- **Encapsulation:**
  - Here's an example of encapsulation that hopefully makes things more clear:


  - Here we have an Arduino Uno, and an Arduino Uno within an enclosure. **An enclosure is a great representation of what encapsulation is all about**
  - Encapsulation aims to protect certain components from outside influences and knowledge as well as expose components which other things should interface with. In programming terms, this involves **information hiding** though **access modifiers**, which change the extent to which certain variables and/or properties can be read and written
  - But beyond that, encapsulation also aims to provide those external interfaces much more effectively. With our Arduino example, this could include the nice buttons and screen which makes the user's interaction with the device much simpler. They provide the user with simple ways to affect the **device's behavior** and gain useful information about its operation which would otherwise be much more difficult
    - In programming, this involves the grouping of various components into a separable construct, such as a function, class, or object. It also includes providing the means of interacting with those constructs, as well as methods for gaining useful information about them
  - Encapsulation helps programmers in many many additional ways, not least of which is improved code maintainability and testability

- **Abstraction:**
  - Generalization is actually a specific *type of* abstraction, not the other way around. In other words, all generalizations are abstractions, but all abstractions are *not* necessarily generalizations
    - Here's how I like to think of abstraction:

    - Would you say the image there is a tree? Chances are you would. But is it *really* a tree? Well, of course not! It's a bunch of pixels made to look like something we might call a tree. We could say that it represents an abstraction of a real tree. Notice that several visual details of the tree are omitted. Also, it does not grow, consume water, or produce oxygen. How could it? it's just a bunch of colors on a screen, represented by bytes in your computer memory
    - And **here is the essence of abstraction. It's a way of simplifying things so they are easier to understand**. **Every idea going through your head is an abstraction of reality**. Your mental image of a tree is no more an actual tree than this jpeg is
    - In programming, we might use this to our advantage by creating a Tree class with methods for simulated growing, water consuming, and oxygen production. Our creation would be something that represents our experience of actual trees, and only includes those elements that we really care about for our particular simulation. We use abstraction as a way of representing our experience of something with bytes and mathematics

➢ **Abstract Classes:**

- Abstraction in programming also allows us to consider commonalities between several "concrete" object types (types that actually exist) and define those commonalities within a unique entity. For example, our Tree class may inherit from an abstract class Plant, which has several properties and methods which are applicable to all of our plant-like classes, but *removes* those that are specific to each type of plant. This can significantly reduce duplication of code, and improves maintainability
- **The practical difference of an abstract class and plain class is that conceptually there's no "real" instances of the abstract class.** It wouldn't make sense to construct a Plant object because that's not specific enough. Every "real" Plant is also a more specific type of Plant
- Also, if we want our program to be more realistic, we might want to consider the fact that our Tree class might be too abstract itself. In reality, every Tree is a more specific type of Tree, so we could create classes for those types such as Birch, Maple, etc. which inherit from our, perhaps now abstract, Tree class

JVM:

- Another good example of abstraction is the **Java Virtual Machine (JVM)**, which provides a virtual or abstract computer for Java code to run on. It essentially takes away all of the platform specific components of a system, and provides an abstract interface of "computer" without regard to any system in particular

➢ **The Difference:**

- Encapsulation differs from abstraction in that it doesn't have anything to do with how 'real' or 'accurate' something is. It doesn't *remove* components of something to make it simpler or more widely applicable, Rather it may *hide* certain components to achieve a similar purpose.

Abstraction:

- Ignoring those aspects of an object that are not relevant to the current scope of the problem
- Reduces scope and helps managing complexity
- Exclude which is not relevant to the current scope

Encapsulation:

- Provide interface to access the functionality of the object & hides how it is implemented
- Keep the attributes and behavior as one unit-helps to make it more independent

➢ Conclusion:

- Abstraction comes before encapsulation
- In abstraction, we are trying to come up with the scope of the problem
- In Encapsulation, we are trying to come up with the approach to solve and implement problem in a better way

II. Abstraction:

➢ **Think about Abstract like a template, an idea, like a ghost**

```
# Prevents a user from creating an object of that class and compels the user to implement any abstract methods found   within the child class
# compels a user to override abstract methods in a child
# abc = abstract base class
# transformation will be done in 3 steps (import,(ABC), decorator)
#abstract class = a class which contains one or more abstract methods
# abstract method = a method that has a declaration but does not have implementation

# Think about abstract like a template, an idea, like a ghost

# to prevent user from creating an object of the class Vehicle is by turning the class into abstract class

# ex. let us pretend that we are creating a need for sped game and we need
 # for the user to create an object from a specific kind of vehicle whether car from the car class,
 # or motorcycle from the motorcycle class, we would like the user to be prevented to create an object of
 # the vehicle class because the vehicle class is too generic and does not have all implementation set up
 # for a vehicle it would be like somebody unlocked the invisible car which it does not exist and missing
 # a lot of features, we need for the user to create an object from the child class because those are already
```

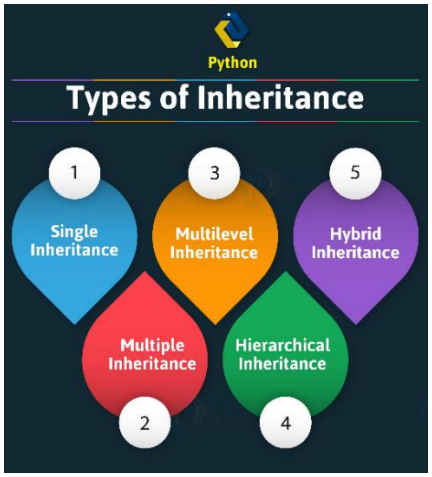
```
 # flushed out, so one way of preventing the user of preventing user to create object of the Vehicle class
 # is to turn it into an abstract class
 # in the moment in which your class transforms into an abstract class you'll not be able to create an object of that class
 # declaring simply mean it does not have a body and you can do it through pass
 # pass means I don't have anything inside this method

from abc import ABC, abstractmethod # step one
class Vehicle(ABC): # step two put in parentheses ABS for the class you want to inherit from ABC class
                    # once you add abstractmethod to the class you will be prevented to create a vehicle object
                      # because our vehicle class is now an abstract class, and we cannot give it a physical form, a physical manifestation
  @abstractmethod # step three add decorator
  def go(self):
  #This method has declaration but not implementation
    """we are just defining it not implementing the go method"""
    pass

  @abstractmethod # so we will have to override this and implement by child class
  def stop(self):
    pass

class Car(Vehicle):
  def go(self):
  # will be forced to implement and override parent class otherwise this class will not be allowed to be instantiated
    """we are overriding the go method and creating our own implementation """
    print("You drive the car")
class Motorcycle(Vehicle):
  def go(self):
    print("You ride the motorcycle")
class Byke(Vehicle):
  pass #Can't instantiate abstract class Vehicle with abstract method go
     # So abstract class will force us to implement

vehicle = Vehicle() # Can't instantiate abstract class Vehicle with abstract method go
car = Car()
motorcycle = Motorcycle()
byke = Byke()

vehicle.go()
car.go()
motorcycle.go()

# Cz: so our vehicle class is telling its children, if you will inherit from me
      # then you need to override this abstract method of mine and if you don't
      # I'm NOT going to let you be instantiated it so in order to create a car and motorcycle class we need to override
      # the go method that they inherit from its parents class of Vehicle and provide its own implementation
```

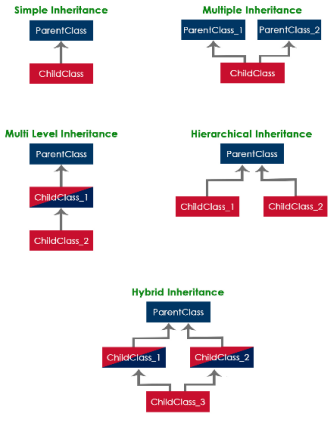
- *Encapsulation: --* **Information hiding**
- *Abstraction: --* **Implementation hiding**

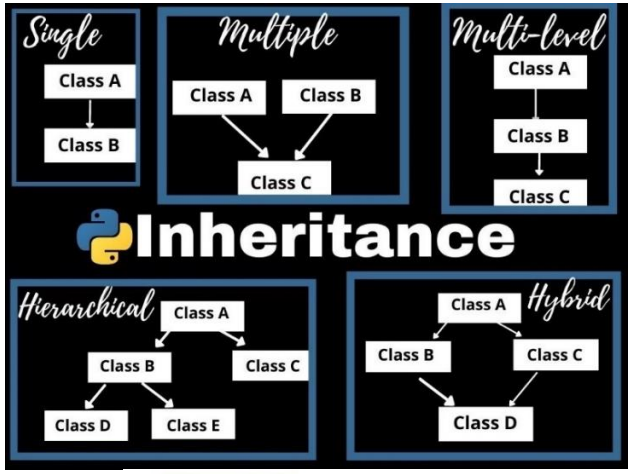
This is ubiquitous in the code, we write from a user clicking a button, and remaining blissfully unaware of event listeners waiting to invoke callback functions, all the way down to the gems and libraries we use ourselves. These are themselves abstractions of functionality we need to use but do not need to build from scratch. It reduces complexity and increases efficiency by making things more intuitive, modular and overall less complicated

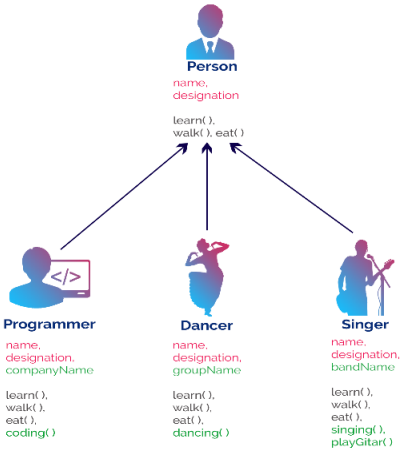
Based on our person class, we can see that some of the data and functionality has been abstracted. No longer do we have to call "first_name" and "last_name" on our person instance but rather can simply ask the object who it is or how old it is, It returns nicely formatted strings that interpolate the data we have stored or data we have calculated. The user does not need to see this, they only need a simple interface and a result. Additionally, we make our attr readers and age method private to further embody the encapsulation principle and protect access to data the user does not require

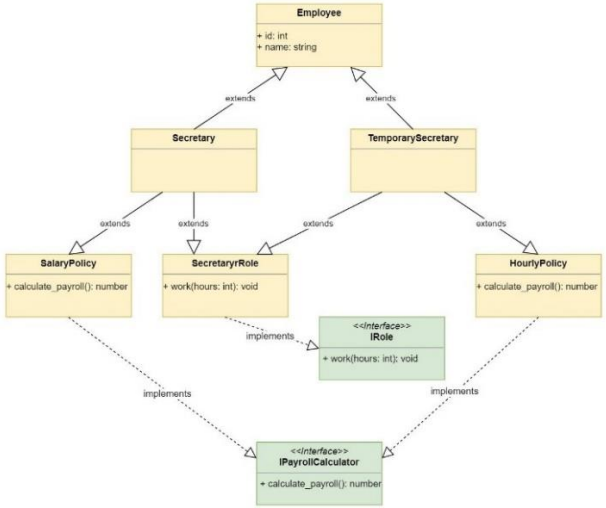
III. Inheritance:

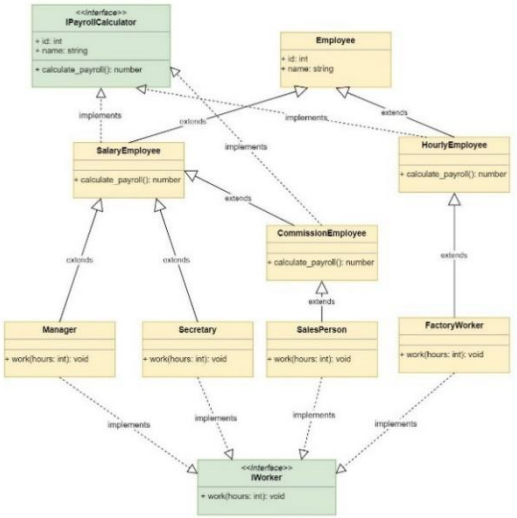












- The **process of inheriting the properties of the parent class into a child class is called inheritance**
- The existing class is called a **base class or parent class** and the new class is called a **subclass or child class or derived class**
- In Object-oriented programming, inheritance is an important aspect. The main purpose of inheritance is the **reusability** of code because **we can use the existing class to create a new class instead of creating it from scratch**
- The Python super() function returns objects represented in the parent's class and is very useful in multiple and multilevel inheritances to find which class the child class is extending first
- In inheritance, the child class acquires all the data members, properties, and functions from the parent class. Also, a child class can also provide its specific implementation to the methods of the parent class
  - **For example**, In the real world, Car is a sub-class of a Vehicle class. We can create a Car by inheriting the properties of a Vehicle such as Wheels, Colors, Fuel tank, engine, and add extra properties in Car as required
    - **Syntax:**

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

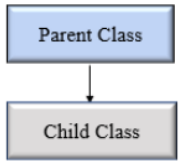
III.A. Types Of Inheritance (single parent):

- In Python, **based upon the number of child and parent classes involved**, there are **five types** of inheritance:
  1. **Single inheritance (single parent with a child)**
  2. **Multiple Inheritance (2 parents with a child)**
  3. **Multilevel inheritance (grandparent with son and grandson)**

4. **Hierarchical Inheritance (single parent with multiple children)**
5. **Hybrid Inheritance – combination of above**

III.A.1. Single Inheritance:

- In single inheritance, a child class inherits from a single-parent class. Here is one child class and one parent class:



- **Example:**
  - Let's create one parent class called Vehicle and one child class called Car to implement single inheritance

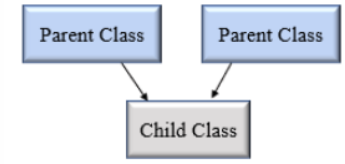
```
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Create object of Car
car = Car()

# access Vehicle's info using car object
car.Vehicle_info()
car.car_info()
```

Output

```
Inside Vehicle class
Inside Car class
```

III.A.2. Multiple Inheritance (2 parents):

- In multiple inheritance, one child class can inherit from multiple parent classes. So here is one child class and multiple parent classes



- **Example:**

```
# Parent class 1
class Person:
    def person_info(self, name, age):
        print('Inside Person class')
        print('Name:', name, 'Age:', age)

# Parent class 2
class Company:
    def company_info(self, company_name, location):
```

```
        print('Inside Company class')
        print('Name:', company_name, 'location:', location)

# Child class
class Employee(Person, Company):
    def Employee_info(self, salary, skill):
        print('Inside Employee class')
        print('Salary:', salary, 'Skill:', skill)

# Create object of Employee
emp = Employee()

# access data
emp.person_info('Jessa', 28)
emp.company_info('Google', 'Atlanta')
emp.Employee_info(12000, 'Machine Learning')
```

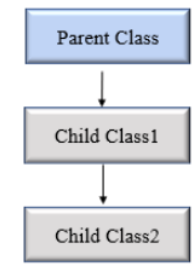
Output

```
Inside Person class
Name: Jessa Age: 28

Inside Company class
Name: Google location: Atlanta

Inside Employee class
Salary: 12000 Skill: Machine Learning
```

- In the above example, we created two parent classes `Person` and `Company` respectively. Then we create one child called `Employee` which inherit from Person and Company classes

III.A.3. Multilevel inheritance (grandparent):

- In multilevel inheritance, a class inherits from a child class or derived class. Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B
- In other words, we can say a **chain of classes** is **called multilevel inheritance**



Example:

```
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Child class
class SportsCar(Car):
    def sports_car_info(self):
        print('Inside SportsCar class')

# Create object of SportsCar
```

```
s_car = SportsCar()

# access Vehicle's and Car info using SportsCar object
s_car.Vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

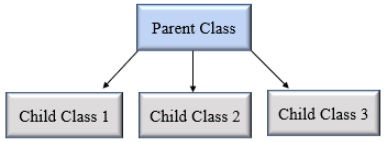
Output

```
Inside Vehicle class
Inside Car class
Inside SportsCar class
```

- In the above example, we can see there are three classes named Vehicle, Car, SportsCar. Vehicle is the superclass, Car is a child of Vehicle, SportsCar is a child of Car. So we can see the **chaining of classes**

III.A.4. Hierarchical Inheritance (single parent with multiple children):

- In Hierarchical inheritance, more than one child class is derived from a single parent class. In other words, we can say one parent class and multiple child classes



Example:

- Let's create 'Vehicle' as a parent class and two child class 'Car' and 'Truck' as a parent class

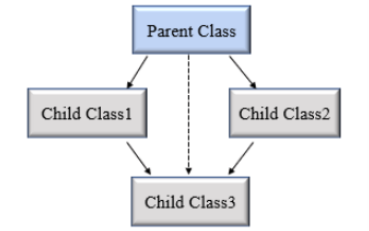
```
class Vehicle:
    def info(self):
        print("This is Vehicle")

class Car(Vehicle):
    def car_info(self, name):
        print("Car name is:", name)

class Truck(Vehicle):
    def truck_info(self, name):
        print("Truck name is:", name)

obj1 = Car()
obj1.info()
obj1.car_info('BMW')

obj2 = Truck()
obj2.info()
obj2.truck_info('Ford')
```

Output

```
This is Vehicle
Car name is: BMW

This is Vehicle
Truck name is: Ford
```

III.A.5. Hybrid Inheritance (combination of above):

- When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance



- **Example:**

```
class Vehicle:
    def vehicle_info(self):
        print("Inside Vehicle class")

class Car(Vehicle):
    def car_info(self):
        print("Inside Car class")

class Truck(Vehicle):
    def truck_info(self):
        print("Inside Truck class")

# Sports Car can inherits properties of Vehicle and Car
class SportsCar(Car, Vehicle):
    def sports_car_info(self):
        print("Inside SportsCar class")

# create object
s_car = SportsCar()

s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

- **Note**: In the above example, **hierarchical** and **multiple** inheritance exists. Here we created, parent class Vehicle and two child classes named Car and Truck this is hierarchical inheritance
  - Another is SportsCar inherit from two parent classes named Car and Vehicle. This is multiple inheritance

III.B. Python super() function:

- When a class inherits all properties and behavior from the parent class is called inheritance. In such a case, the inherited class is a subclass and the latter class is the parent class
  - **In child class, we can refer to parent class by using the super() function.** The super function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method
- **Benefits of using the super() function:**
  1. We are not required to remember or specify the parent class name to access its methods.
  2. We can use the super() function in both **single** and **multiple inheritances**.
  3. The super() function support code **reusability** as there is no need to write the entire function

- **Example:**

```
class Company:
    def company_name(self):
        return 'Google'

class Employee(Company):
    def info(self):
        # Calling the superclass method using super()function
        c_name = super().company_name()
        print("Jessa works at", c_name)

# Creating object of child class
emp = Employee()
emp.info()
```

Output:

```
Jessa works at Google
```

- In the above example, we create a parent class `Company` and child class `Employee`. In `Employee` class, we call the parent class method by using a `super()` function

III.B.1. Example of super() function in Python:

```
class Emp():
    """ Example 1 """
    def __init__(self, id, name, Add):
        self.id = id
        self.name = name
        self.Add = Add

# Class freelancer inherits EMP
class Freelance(Emp):
    def __init__(self, id, name, Add, Emails):# first you initialize then you use super to access the attributes, check if all parameters activated by having what was in the parent class in the child __init__
                                              and in super without self and the parameter of the child class SO super will contain ONLY the parent attributes without self
        super().__init__(id, name, Add)
        self.Emails = Emails

Emp_1 = Freelance(103, "Suraj kr gupta", "Noida" , "KKK@gmails")
print('The ID is:', Emp_1.id)
print('The Name is:', Emp_1.name)
print('The Address is:', Emp_1.Add)
print('The Emails is:', Emp_1.Emails)
```

Output:

```
The ID is: 103
The Name is: Suraj kr gupta
The Address is: Noida
The Emails is: KKK@gmails
```

- Understanding Python super() with __init__() methods
- Python has a reserved method called "__init__." In Object-Oriented Programming, it is referred to as a constructor. When this method is called it allows the class to initialize the attributes of the class
- In an inherited subclass, a parent class can be referred with the use of the super() function. The super function returns a temporary object of the superclass that allows access to all of its methods to its child class

```
class Animals:
    """ Example 2 """
    # Initializing constructor
    def __init__(self):
        self.legs = 4
        self.domestic = True
        self.tail = True
        self.mammals = True

    def isMammal(self):
        if self.mammals:
            print("It is a mammal.")

    def isDomestic(self):
        if self.domestic:
            print("It is a domestic animal.")

class Dogs(Animals):
    def __init__(self):
        super().__init__()

    def isMammal(self):
        super().isMammal()
```

```
class Horses(Animals):
  def __init__(self):
    super().__init__()

  def hasTailandLegs(self):
    if self.tail and self.legs == 4:
      print("Has legs and tail")

# Driver code
Tom = Dogs()
Tom.isMammal()
Bruno = Horses()
Bruno.hasTailandLegs()
```

Output:
It is a mammal
Has legs and tail

- **Super function in multiple inheritances**
- Let's take another **example of a super function**, Suppose a class **canfly** and **canswim** inherit from a mammal class and these classes are inherited by the animal class
- So the animal class inherits from the multiple base classes. Let's see the use of **Python super with arguments** in this case

```
class Mammal():
  """ Example 3 """
  def __init__(self, name):
    print(name, "Is a mammal")

class canFly(Mammal):

  def __init__(self, canFly_name):
    print(canFly_name, "cannot fly")

    # Calling Parent class
    # Constructor
    super().__init__(canFly_name)

class canSwim(Mammal):

  def __init__(self, canSwim_name):

    print(canSwim_name, "cannot swim")

    super().__init__(canSwim_name)

class Animal(canFly, canSwim):

  def __init__(self, name):
    super().__init__(name)

# Driver Code
Carol = Animal("Dog")
```

Output:

- **The class Animal inherits from two-parent classes – canFly and canSwim. So, the subclass instance Carol can access both of the parent class constructors**

  Dog cannot fly
  Dog cannot swim
  Dog Is a mammal

- **MRO in Multiple Inheritance**

```
class A:
    """ Example 4 """
    def age(self):
        print("Age is 21")
class B:
    def age(self):
        print("Age is 23")
class C(A, B):
    def age(self):
        super(C, self).age()

c = C()
print(C.__mro__)
print(C.mro())
```

Output:

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

- Multi-Level inheritance
- Let's take another **example of a super function**, suppose a class canSwim is inherited by canFly, canFly from mammal class
- So the mammal class inherits from the Multi-Level inheritance. Let's see the use of **Python super with arguments** in this case

```
class Mammal():
    """ Example 5 """
    def __init__(self, name):
        print(name, "Is a mammal")


class canFly(Mammal):

    def __init__(self, canFly_name):
        print(canFly_name, "cannot fly")

        # Calling Parent class
        # Constructor
        super().__init__(canFly_name)


class canSwim(canFly):

    def __init__(self, canSwim_name):

        print(canSwim_name, "cannot swim")

        super().__init__(canSwim_name)


class Animal(canSwim):

    def __init__(self, name):

        # Calling the constructor
        # of both the parent
        # class in the order of
        # their inheritance
        super().__init__(name)


# Driver Code
Carol = Animal("Dog")
```

Output:

```
Dog cannot swim
Dog cannot fly
Dog Is a mammal
```

```python
class Company:
    """ Example 6"""
    def __init__(self, name):
        self.name = name

    def company_name(self):
        return 'Google'

class Employee(Company):
    def info(self, name):
        """ using super to access the attribute
         and method of parent class"""
        # super to access the parent method
        c_name = super().company_name()
        print("Jessa works at", c_name)
        # super to access and change the parent attribute
        super().__init__(name)

emp = Employee('First argument')
print(emp.name) # prints arguments received from the parent class
emp.info('Second Argument') #Jessa works at Google
print(emp.name) # prints arguments received from child class
```

```python
# super() = Function used to give access to the methods of a parent class.
# Return object of a parent class when used

class Rectangle:
    def __init__(self, length, width): # so Square and Cube had same as below => we write in the parent then we access it through super
        self.length=length
        self.width=width

class Square(Rectangle):
    def __init__(self, length, width):
        super().__init__(length,width)

    def area(self):
        return self.length*self.width

class Cube(Rectangle):
    def __init__(self, length,width, height):
        super().__init__(length,width)
        self.height=height

    def volume(self):
        return self.length*self.width*self.height

square=Square(3, 3)
cube = Cube(3,3,3)

print(square.area())
print(cube.volume()) # 9 and 27
```

III.B.2. The benefits of using a super() function are:

- Need not remember or specify the parent class name to access its methods. This function can be used both in single and multip le inheritances
- You can use it to access a method in the parent or attributes using __init__

- This implements modularity (isolating changes) and code reusability as there is no need to rewrite the entire function
- Super function in Python is called dynamically because Python is a dynamic language, unlike other languages
- Super function in single inheritance, Let's take the example of animals. Dogs, cats, and cows are part of animals. They also share common characteristics like:
  - They are mammals
  - They have a tail and four legs
  - They are domestic animals
    - So, the classes dogs, cats, and horses are a subclass of animal class. This is an example of single inheritance because many subclasses is inherited from a single parent class

III.C. issubclass():

- In Python, we can verify whether a particular class is a subclass of another class. For this purpose, we can use Python built-in function issubclass()
- This function returns True if the given class is the subclass of the specified class. Otherwise, it returns False

▸ **Syntax**

```
issubclass(class, classinfo)
```

▸ **Where,**

- class: class to be checked
- classinfo: a class, type, or a tuple of classes or data types

▸ **Example:**

```
class Company:
    def fun1(self):
        print("Inside parent class")

class Employee(Company):
    def fun2(self):
        print("Inside child class.")

class Player:
    def fun3(self):
        print("Inside Player class.")

# Result True
print(issubclass(Employee, Company))

# Result False
print(issubclass(Employee, list))

# Result False
print(issubclass(Player, Company))

# Result True
print(issubclass(Employee, (list, Company)))

# Result True
print(issubclass(Company, (list, Company)))
```

III.D. Method Resolution Order in Python (MRO):

- In Python, Method Resolution Order(MRO) is the order by which **Python looks for a method or attribute**
- First, the method or attribute is searched within a class, and then it follows the order we specified while inheriting
- This order is also called the Linearization of a class, and a set of rules is called **MRO (Method Resolution Order)**
- The **MRO plays an essential role in multiple inheritances as a single method may found in multiple parent classes**,
- In multiple inheritance, the following search order is followed:

▸ So below refers if the method name is the same and the order python looks for it, if the method name is the same you need to make sure you do not put the parent name first as it will not work

1. First, it searches in the current parent class if not available, then searches in the parents class specified while inheriting (that is left to right)
2. We can get the MRO of a class. For this purpose, we can use either the mro attribute or the mro() method

- **Example:**

```
class A:
    def process(self):
        print(" In class A")

class B(A):
    def process(self):
        print(" In class B")

class C(B, A):
    def process(self):
        print(" In class C")

# Creating object of C class
C1 = C()
C1.process()
print(C.mro())
# In class C
# [<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

  - In the above example, we create three classes named A, B and C. Class B is inherited from A, class C inherits from B and A
    - When we create an object of the C class and calling the process() method, Python looks for the process() method in the current class in the C class itself
      - Then search for parent classes, namely B and A, because C class inherit from B and A. that is, C(B, A) and always search in **left to right manner**

III.E. Method Chaining:

- **method chaining = calling multiple methods sequentially**
- **Each call performs an action on the same object and returns self**

```
class Car:
    def turn_on(self):
        print("You start the engine")
        return  self # you have to add this for Method Chaining
    def drive(self):
        print("You drive the car")
        return self
    def brake(self):
        print("You step on the brakes")
        return self
    def turn_off(self):
        print("You turn off the engine")
        return self

car=Car()
car.turn_on()\
    .drive()\
    .brake()\
    .turn_off()
# \ is line continuation
```

Inheritance:

- Refers to defining a new class with little or no modification to an existing class
- A ***sub-class*** is derived from a ***base-class***, inheriting its behavior, and making behavior specific to sub-class
- Inheritance allows a derived class to inherit all the features from its base class, adding new features to it. This results in ***re-usability of code***

Syntax

```
# Base class
class BaseClass:
   Body of base class

# Derived class
class DerivedClass(BaseClass):
   Body of derived class
```

Example1: inheritance - used for reusing the code, below simple example:

```
class Mamal:
  def walk(self):
    print("walk")

class Dog(Mamal):
  def bark(self):
    print("bark")

class Cat(Mamal):
  def be_annoying(self):
    print("annoying")

dog1 = Dog()
dog1.walk()
dog1.bark()

cat1 = Cat()
cat1.be_annoying()
```

↥ use pass if there is nothing in the method so python will not give error

Example2: 2 different kinds of houses and create a class for each one of them:

```
class Apartment:
  '''
  A house within a large building along with other houses   '''

  def __init__(self, rooms, bathrooms, floor):
     self.rooms = rooms
     self.bathrooms = bathrooms
     self.floor = floor

  def room_details(self):
     print(f'This property has {self.rooms} rooms \
          with {self.bathrooms} bathrooms')


class Bungalow:
  '''
  A (typically) one-story landed house
  '''
```

```
  def __init__(self, rooms, bathrooms):
      self.rooms = rooms
      self.bathrooms = bathrooms

  def room_details(self):
      print(f'This property has {self.rooms} rooms \
          with {self.bathrooms} bathrooms')
```

- As we can easily observe — both ***Apartment*** and ***Bungalow*** are kind of houses and have some common properties (rooms, bathrooms) and also common behaviour(room_details)
- Currently, these common properties and behavior are being duplicated in both the classes, which can easily be extracted out in order to be re-used. This is where the concept of **Inheritance** can be of help

Example3: we can create a Base class with all the common properties and reuse the same in the base classes

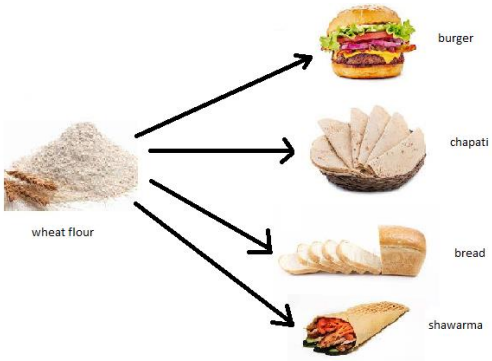
```
# Base class
class House:
    '''
    A place which provides with shelter or accommodation
    '''

    def __init__(self, rooms, bathrooms):
        self.rooms = rooms
        self.bathrooms = bathrooms

    def room_details(self):
        print(f'This property has {self.rooms} rooms \
            with {self.bathrooms} bathrooms')

class Apartment(House):
    '''
    A house within a large building where others also have
    their own house
    '''

    def __init__(self, rooms, bathrooms, floor):
        House.__init__(self, rooms, bathrooms)
        self.floor = floor

class Bungalow(House):
    '''
    A (typically) one-story landed house
    '''
    pass

# Create an Apartment
apartment = Apartment(2, 2, 21)
apartment.room_details()

# Create a Bungalow
bungalow = Bungalow(4, 3)
bungalow.room_details()

Output:
This property has 2 rooms with 2 bathrooms
This property has 4 rooms with 3 bathrooms
```

- We have successfully created the House base-class with the common properties. Both ***Apartment*** and ***Bungalow*** now extends the ***House*** class. This makes our code neat, maintainable, and reusable

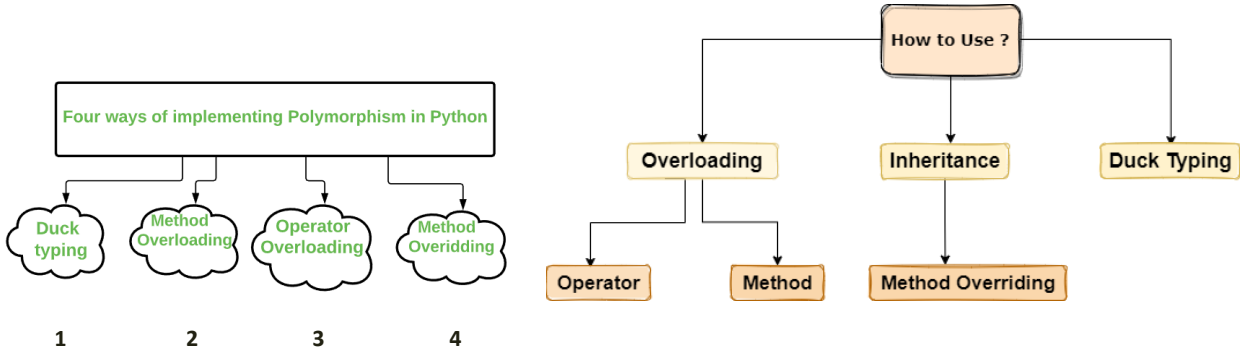
IV. Polymorphism:

- **Polymorphism is quite a famous and interesting topic in programming. As a word, it means occurring in several different forms**
  - **In programming, it can be termed as a function or combination of functions having the same name but different functionality**
- Polymorphism is a word that came from two Greek words, **poly** means many and **morphos** means forms. If something exhibits various forms, it is called polymorphism

➢ Let's take a simple example in our daily life. Assume that we have wheat flour. Using this wheat flour, we can make burgers, chapati, shawarma or loaves of bread
-> It means same wheat flour is taking different edible forms and hence we can say wheat flour is exhibiting polymorphism. Consider Figure:



Polymorphism where wheat flour takes different edible forms



- A variable may store different types of data, an object may exhibit different behaviors in different contexts, or a method may perform various tasks in Python. This type of behavior is called polymorphism
  - So, how can we define polymorphism? **If a variable, object or method exhibits different behavior in different contexts, it is called polymorphism. Python has built-in polymorphism**
- So, for polymorphism simply get many(poly) forms(morph). Back to the auto lot:
  - Say we have this vintage civic:

And this Tesla Model X:

- When we sit in the driver's seat and push on the gas both cars go. But under the hood one is burning gas and the other runs on electricity
  - To the user both cars will move forward, but inside the engine different things are happening. This is the crux of polymorphism
  - Functions or methods that are named the same thing can do things differently. This makes our code easier to interact with
  - If we look back to our Person and Child classes:
    - Children are not adults. Perhaps we can reflect that in our code. When you ask a child who they are, they may be a little more informal
    - They might overshare. If we override the 'who_am_i?" method in the child class to only offer a first name but additionally offer an age, we still have not changed the interface with the code from the user perspective
    - We can call that method on an instance of the Person class or the Child class and both will respond
    - They will do so through different implementations and the output will be slightly different but to a user both work. It is polymorphic
  - A last quick note about polymorphism is that, specifically in inheritance patterns such as this, one of the fundamentally useful aspects of this principle is that affecting or overriding a method in the child class does not change that same method in the parent class. This can be incredibly useful in creating more specialized behaviors down the inheritance chain

- Polymorphism can be broadly categorized into **two types**:

IV.A. Inbuilt Polymorphic Functions:

- **These are the built-in operators or functions that can be used in different ways**
- In the code snippet below, we can see how the addition operator can be used to add two integers, concatenate two strings, and add two lists as well
- Similarly, we can see how the built-in function `len()` can be used to detect length of a string and lists too

Example of built-in Polymorphic Functions:

```
print("Output")
print("Poly" + " " + "Morphism")
print(5 + 6)
print(["Poly", 2, "3"] + ["4", 5, "Morphism"])

print(len("PolyMorphism"))
print(len([1, 2, 3, 4, 5, 6]))
```

```
Output
Poly Morphism
11
['Poly', 2, '3', '4', 5, 'Morphism']
12
6
```

IV.B. User-Defined Polymorphic Functions:

- These, as the name suggests, are the functions that are defined or written by a user to perform different functionalities
- In the code snippet below, we can use the multiply function to multiply two numbers, three numbers, or four numbers. We can modify this function according to our needs, and thus it can serve us in multiple ways

Example of User-Defined Functions:

```
def multiply(num1, num2, num3=1, num4=1):
   return num1 * num2 * num3 * num4

print("Output")
final_mul = multiply(2, 3)
print(final_mul)
final_mul = multiply(2, 3, 4)
print(final_mul)
final_mul = multiply(2, 3, 4, 5)
print(final_mul)
```

```
Output
6
24
120
```

The following topics are examples for polymorphism in Python:

IV.1. Operator Overloading
IV.2. Method Overloading
IV.3. Method Overriding
IV.4. Duck Typing Philosophy of Python

IV.1. Operator overloading:

- **Modifying the behavior of an operator by redefining the method of an operator invoked is called Operator Overloading,**
  It allows operators to have extended behavior beyond their pre-defined behavior depending on the operands (values) that we use, in other words, we can use the same operator for multiple purposes
- For example, the + operator will perform an arithmetic addition operation when used with numbers. Likewise, it will perform concatenation when used with strings
- The operator + is used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python
  - So, basically **defining methods for operators is known as operator overloading**. For e.g: **To use the + operator with custom objects you need to define a method called __add__**
- Special symbols in Python that we use to perform various operations on objects are called Operators. The objects on which operations are performed are called Operands

- We say objects because everything is an object in Python

- **Example**:

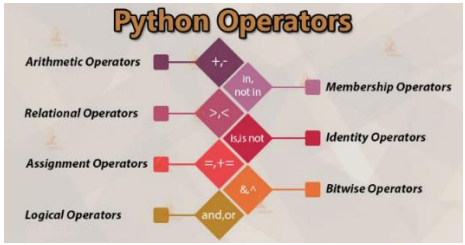
```
# add 2 numbers
print(100 + 200)

# concatenate two strings
print('Jess' + 'Roy')

# merger two list
print([10, 20, 30] + ['jessa', 'emma', 'kelly'])
```

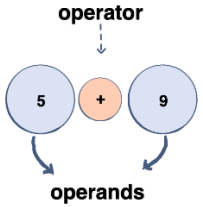
Output:

```
300
JessRoy
[10, 20, 30, 'jessa', 'emma', 'kelly']
```

IV.1.1. Types of Operators in Python:

- **There are seven types of operators in Python:**



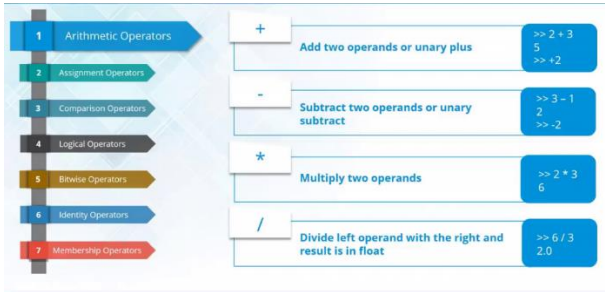


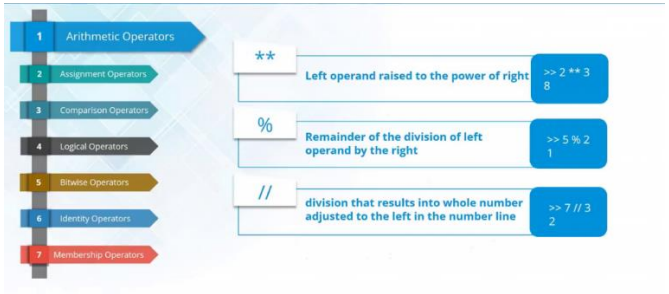
- **Among those, we can overload 5 types of operators:**

- **NO Logical and Identity Operators**

➢ ***Types of operators in Python we can overload***:

IV.1.1.1. Arithmetic operators:





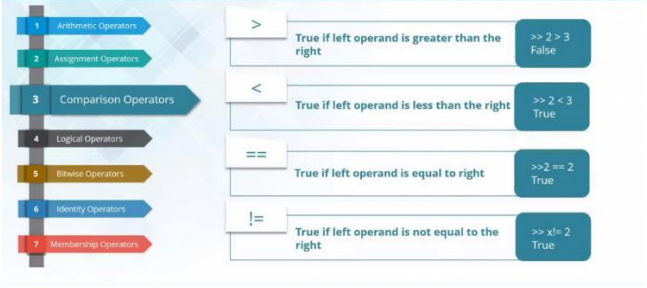

```
# Output: x ** y = 5062
```

```
print('x ** y =', x**y)

# Output: x % y = 3
print('x % y =', x%y)

# Output: x // y = 3
print('x // y =', x//y)
```

*
**

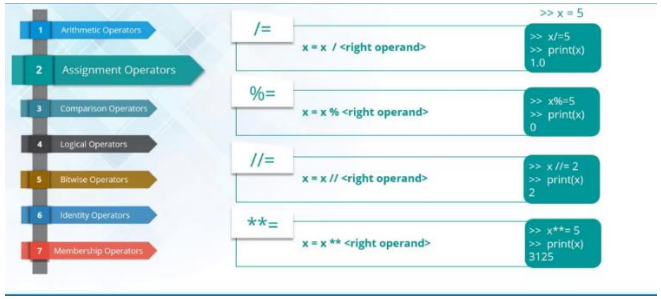
```
>>> 10 / 5
2.0
>>> type(10 / 5)
<class 'float'>

>>> 10 // 4
2
>>> type(10 // 4)
<class 'int'>
```

IV.1.1.2. Comparison operators:

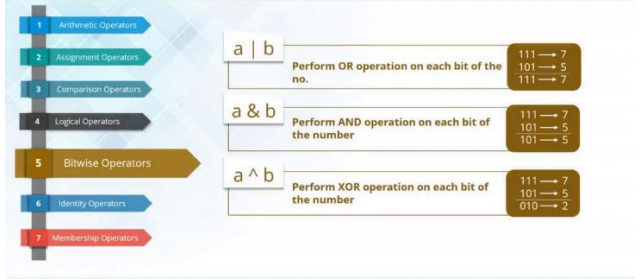


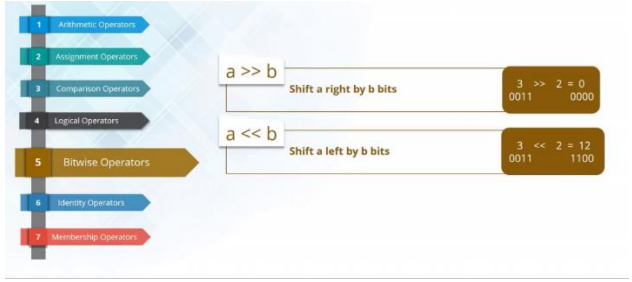

```
x = 10
y = 12

# Output: x > y is False
print('x > y  is', x>y)

# Output: x < y is True
print('x < y  is', x<y)

# Output: x == y is False
print('x == y is', x==y)

# Output: x != y is True
print('x != y is', x!=y)

# Output: x >= y is False
print('x >= y is', x>=y)

# Output: x <= y is True
print('x <= y is', x<=y)
```

IV.1.1.3. Assignment operators:

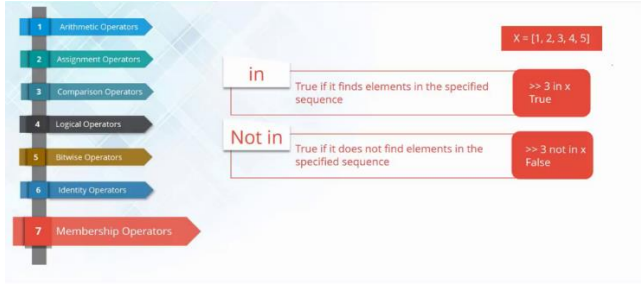


IV.1.1.4. Bitwise operators:





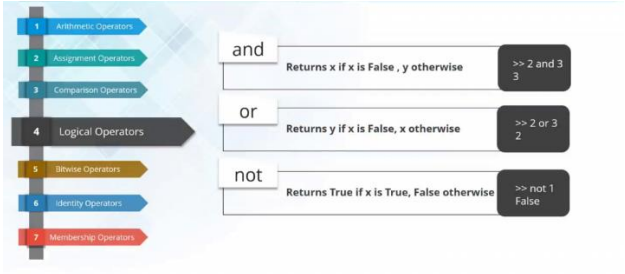
IV.1.1.5. Membership operators:




```
x = 'Hello world'
y = {1:'a', 2:'b'}

# Output: True
print('H' in x)

# Output: True
print('hello' not in x)

# Output: True
print(1 in y)

# Output: False
print('a' in y)
```

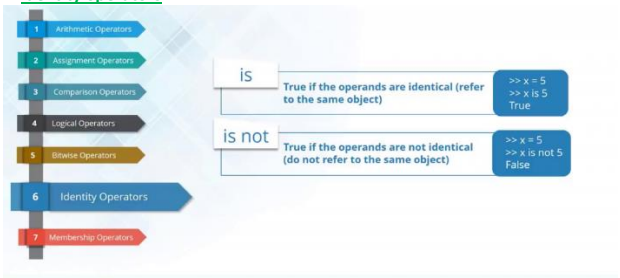
➢ ***Types of Python operators we cannot overload***:

IV.1. Logical operators:




```
x = 5

# Output: x < 10 is True
print('x < 10 is', x < 10)

x = True
y = False

# Output: x and y is False
print('x and y is', x and y)

# Output: x or y is True
print('x or y is', x or y)

# Output: not x is False
print('not x is', not x)
```

IV.2. Identity operators:



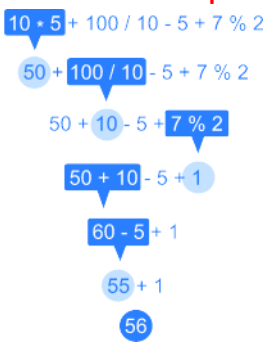

```
>>> x = 5
>>> y = 4 + 1
>>> print(x, y)
5 5

>>> x == y
True
>>> x is y
False
```

```
>>> a = "I am a string"
>>> b = a
>>> id(a)
55993992
>>> id(b)
55993992
```

```
>>> a is b
True
>>> a == b
True
```

```
>>> x = 5
>>> y = 4 + 1
>>> x is not y
True
```

----------Operator Precedence----------



Precedence	Operator Sign	Operator Name
Highest	**	Exponentiation
	+x, -x, ~x	Unary positive, unary negative, bitwise negation
	*, /, //, %	Multiplication, division, floor, division, modulus
	+, -	Addition, subtraction
	<<, >>	Left-shift, right-shift
	&	Bitwise AND
	^	Bitwise XOR
	|	Bitwise OR
	==, !=, <, <=, >, >=, is, is not	Comparison, identity
	not	Boolean NOT
	and	Boolean AND
Lowest	or	Boolean OR

-

IV.1.1.1. Arithmetic operators are two types: Unary operators and Binary Operators:

A. Unary Operators in Python:

Operator	Method
+	__pos__
–	__neg__
~	__invert__

Example: **Negative Operator (-):**

```
class Number:
  def __init__(self, num):
    self.num = num
  def __neg__(self):
    print("Negative Operator")
    return self.num
n1 = Number(5)
print(-n1)
```

Output

```
Negative Operator
5
```

B. Overloading Binary Operators in Python:

Operator	Method
+	__add__
–	__sub__
*	__mul__
/	__truediv__
//	__floordiv__
%	__mod__
**	__pow__

- Suppose we have two objects, and we want to add these two objects with a binary + operator. However, it will throw an error if we perform addition because the compiler doesn't add two objects
  - See the following example for more details:
    - We know + operator is used for adding numbers and at the same time to concatenate strings. It is possible because the + operator is overloaded by both **int class** and **str class**
    - The operators are actually methods defined in respective classes
    - So if you want to use the + operator to add two objects of some user-defined class then you will have to define that behavior yourself and inform Python about that
  - **Example**:

```
class Book:
    def __init__(self, pages):
        self.pages = pages

# creating two objects
b1 = Book(400)
b2 = Book(300)

# add two objects
print(b1 + b2)
```

Output

```
TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

Overloading the + Operator:

- We can overload + operator to work with custom objects also. Python provides some special or **magic function that is automatically invoked when associated with that particular operator**
  - For example, **when we use the + operator, the magic method __add__() is automatically invoked**. Internally + operator is implemented by using __add__() method
  - We have to override this method in our class if you want to add two custom objects
- **Example1**:

```
class Book:
    def __init__(self, pages):
        self.pages = pages

    # Overloading + operator with magic method
    def __add__(self, other):
        return self.pages + other.pages

b1 = Book(400)
b2 = Book(300)
print("Total number of pages: ", b1 + b2)
```

Output

```
Total number of pages: 700
```

Example2:

```
class Point:
   def __init__(self, x=0, y=0):
      self.x = x
      self.y = y

   def __str__(self):
      return "f{self.x},{self.y}"

   def __add__(self, other):
      x = self.x + other.x
            p1        p1
      y = self.y + other.y
            p2        p2
      return (x,y)

p1 = Point(1,2)
p2 = Point(2,3)

print(p1+p2) # (3,5)
```

- p1+p2 # p1 becomes self and p2 becomes other
- We add return to __add__
- p1.__add__(p2) which in turn is object._add_(self,other) and in our case will be Point.__add__(p1,p2)

Example3:

```
class Books:
   def __init__(self, price):
      self.price = price

   def __add__(self, book):
      return self.price + book.price
           maths.price  science.price
      -> self for the 2nd object will be actually the word "book"
      -> price will be injected from the argument from the objects

maths = Books(250)
science = Books (300)
print("Total Cost:", (maths + science))
              self.price + book.price
   maths becomes self & science becomes book
```

Example4:

```
#overloading + operator to act on objects
class BookX:
  def __init__(self, pages):
     self.pages = pages
  def __add__(self, other):
     return self.pages+other.pages
class BookY:
  def __init__(self, pages):
     self.pages = pages
b1 = BookX(100)
b2 = BookY(150)
print('Total pages=', b1+b2)
```

Example5:

```
class Student:

   # defining init method for class
   def __init__(self, m1, m2):
      self.m1 = m1
      self.m2 = m2
```

```
  # overloading the + operator
  def __add__(self, other):
      m1 = self.m1 + other.m1
      m2 = self.m2 + other.m2
      s3 = Student(m1, m2)
      return s3


s1 = Student(58, 59)
s2 = Student(60, 65)
s3 = s1 + s2
print(s3.m1)
```

Overloading the * Operator:

- The * operator is used to perform the multiplication
  - Let's see how to overload it to calculate the salary of an employee for a specific period. Internally * operator is implemented by using the __mul__() method
    - **SO YOU INVOKE THE OPERATOR AT THE END BUT IT WILL BEHAVE IN A DIFFERENT WAY AS IT WAS OVERLOADED**
      - **Example1**:

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def __mul__(self, timesheet):
        print('Worked for', timesheet.days, 'days')
        # calculate salary
        return self.salary * timesheet.days


class TimeSheet:
    def __init__(self, name, days):
        self.name = name
        self.days = days


emp = Employee("Jessa", 800)
timesheet = TimeSheet("Jessa", 50)
print("salary is: ", emp * timesheet)
```

Output

```
Worked for 50 days
salary is:  40000
```

- **Example2**:

```
#overloading the * operator
class Employee:
  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
  def __mul__(self, other):
    return self.salary*other.days
class Attendance:
  def __init__(self, name, days):
    self.name = name
    self.days = days
x1 = Employee('Srinu', 500.00)
x2 = Attendance('Srinu', 25)
print('This month salary=', x1*x2)
```

Overloading the Modulus (%) Operator:

```
class Number:
  def __init__(self, num):
    self.num = num
  def __neg__(self):
    print("Negative Operator")
    return self.num
n1 = Number(5)
print(-n1)
```

Output

```
5.0
```

IV. 1.1.2. Comparison Operators:

Operator	Method
>	__gt__
<	__lt__
==	__eq__
>=	__ge__
<=	__le__
!=	__ne__

Overloading greater than (>) operator:

- For this purpose the magic method __gt__() should be overridden
  - **For example:**

```
def __gt__(self, other):
    return self.pages>other.pages
```

- **Example:**

```
#overloading > operator
class Ramayan:
  def __init__(self, pages):
    self.pages = pages
  def __gt__(self, other):
    return self.pages>other.pages
class Mahabharat:
  def __init__(self, pages):
    self.pages = pages
b1 = Ramayan(1000)
b2 = Mahabharat(1500)
if(b1>b2):
  print('Ramayan has more pages')
else:
  print('Mahabharat has more pages')
```

Overloading Equal-to Operator (==):

```
class Strings:
  def __init__(self, str):
    self.str = str
  def __eq__(self, other):
    if len(self.str) == len(other.str):
      return "Equal"
    else:
      return "Not Equal"
word1 = Strings("Mom")
word2 = Strings("Dad")
print(word1 == word2)
```

Output

```
Equal
```

IV.1.1.3. Assignment Operators:

Operator	Method

+=	__iadd__
-=	__isub__
*=	__imul__
/=	__idiv__
//=	__ifloordiv__
%=	__imod__
**=	__ipow__
>>=	__irshift__
<<=	__ilshift__
&=	__iand__
^=	__ixor__
|=	__ior__

- **Overloading Addition Assignment Operator (+=):**

```
class Fruit:
  def __init__(self, num):
    self.num = num
  def __iadd__(self, other):
    new = self.num + 5
    return Fruit(new)
obj = Fruit(5)
print(f"Before addition {obj.num = }")
obj += 1
print(f"After addition {obj.num = } ")
```

Output

```
Before addition obj.num = 5
After addition obj.num = 10
```

IV.1.1.4. Bitwise Operators:

Operator	Method
>>	__rshift__
<<	__lshift__
&	__and__
^	__xor__
|	__or__
~	__invert__

- **Overloading And Operator (&):**

```
class Number:
  def __and__(self, other):
    return "Overloaded '&' Operator"
n1 = Number()
n2 = Number()
print(n1 & n2)
```

Output

```
Overloaded '&' Operator
```

IV.1.1.5. Membership Operators:

Operator	Method
in	__contains__

not in

✹ **Overloading in and not in:**

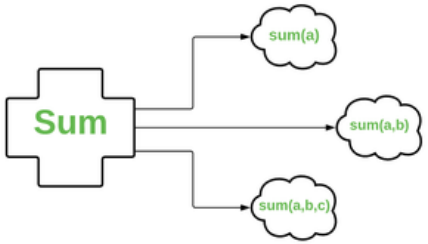
```
class Book:
  def __contains__(self, item):
    print("Book is empty")
    return False
b1 = Book()
b2 = Book()
print(b2 in b1)
print(b2 not in b1)
```

Output

```
Book is empty
False
Book is empty
True
```

IV.2. Method overloading:

- **Method overloading is adding several types of logic to the method to behave differently, as whenever we pass arguments as per our needs, it will operate based upon those**
- If a **method** is written such that it can **perform more than one task**, **it is called method overloading**. We see method overloading in the languages like Java. For example, we call a method as:
  sum(10, 15)
  sum(10, 15, 20)
  - In the first call, we are passing two arguments and in the second call, we are passing three arguments
  - It means, the **sum() method is performing two distinct operations**: finding sum of two numbers or sum of three numbers. **This is called method overloading**
- **So, the simple meaning of method overloading is a class containing multiple methods with the same name but having different parameters**
  - By default, Python does not support method overloading, but we can achieve it by modifying out methods
- **Therefore, the process of calling the same method with different parameters is known as method overloading**
- **Python considers only the latest defined method even if you overload the method. Python will raise a TypeError if you overload the method**
  - Given a single function sum (), the number of parameters can be specified by you. This process of calling the same method in different ways is called **method overloading**



Concept of Method overloading

- **Example1: if we try to use method without overloading, then it will call the second method only**

```
def addition(a, b):
  c = a + b
  print(c)
```

```
def addition(a, b, c):
    d = a + b + c
    print(d)


# the below line shows an error
# addition(4, 5)

# This line will call the second product method
addition(3, 7, 5)
```

- To overcome the above problem, we can use different ways to achieve the method overloading
- **In Python, to overload the class method, we need to write the method's logic so that different code executes inside the function depending on the parameter passes**
- For example, the built-in function range() takes three parameters and produce different result depending upon the number of parameters passed to it

Example:

```
for i in range(5): print(i, end=', ')
print()
for i in range(5, 10): print(i, end=', ')
print()
for i in range(2, 12, 2): print(i, end=', ')
```

Output:

```
0, 1, 2, 3, 4,
5, 6, 7, 8, 9,
2, 4, 6, 8, 10,
```

Example2, Let's understand it with the help of another example:

```
class Book_Food:
    def book(self, lunch = None, dinner = None):
        if lunch and dinner:
            print("Lunch and dinner booked")
        elif lunch:
            print("lunch booked")
        elif dinner:
            print("dinner booked")
        else:
            print("No Food Item booked")

obj = Book_Food()
obj.book()
obj.book(lunch=1)
obj.book(lunch=1,dinner=1)
```

Output:

No Food Item Booked
Lunch Booked
Lunch & Dinner Booked

- In the above example, we can see a class `Book_Food` in which we have a `book` function that has arguments as `None` and whenever we pass arguments it will operate based upon those, so it can behave as a single argument, double argument, and no argument function. We can **add several types of logic** in here to make it operate differently as per our needs

Exmple3, to show method overloading to find sum of two or three numbers:

```
#method overloading
class Myclass:
    def sum(self, a=None, b=None, c=None):
        if a!=None and b!=None and c!=None:
            print('Sum of three=', a+b+c)
        elif a!=None and b!=None:
            print('Sum of two=', a+b)
        else:
            print('Please enter two or three arguments')
#call sum() using object
m = Myclass()
m.sum(10, 15, 20)
```

```
m.sum(10.5, 25.55)
m.sum(100)
```

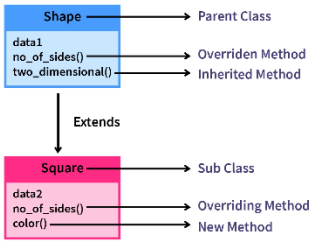
- **Example4, in which** we will assume we have an area() **method to** calculate the area of a square and rectangle. The method will calculate the area depending upon the number of parameters passed to it
  - If one parameter is passed, then the area of a square is calculated
  - If two parameters are passed, then the area of a rectangle is calculated
  - User-defined polymorphic method

```
class Shape:
    # function with two default parameters
    def area(self, a, b=0):
        if b > 0:
            print('Area of Rectangle is:', a * b)
        else:
            print('Area of Square is:', a ** 2)

square = Shape()
square.area(5)

rectangle = Shape()
rectangle.area(5, 3)
```

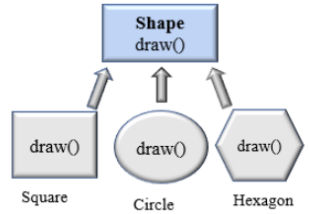
Output:

```
Area of Square is: 25
Area of Rectangle is: 15
```

IV.3. Method Overriding:



- In inheritance, all members available in the parent class are by default available in the child class
- In Python, we can reimplement a method and change its functionality in child class according to our needs. We can inherit methods in the child class from the parent class that have the same name, maybe the same parameters but differ in functionality. This process is known as Method Overriding and is very beneficial in case our parent class method's functionality is not suitable in child class
- If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional functions in the child class. This concept is called **method overriding**
- When a child class method has the same name, same parameters, and same return type as a method in its superclass, then the method in the child is said to **override** the method in the parent class
  - **So if the method name is the same you will use overriding (write same thing and give different value) but if not the same method name then use super()**



- **Example 1:**

```
class Vehicle:
    def max_speed(self):
        print("max speed is 100 Km/Hour")
```

```
class Car(Vehicle):
    # overridden the implementation of Vehicle class
    def max_speed(self):
        print("max speed is 200 Km/Hour")

# Creating object of Car class
car = Car()
car.max_speed()
```

Output:

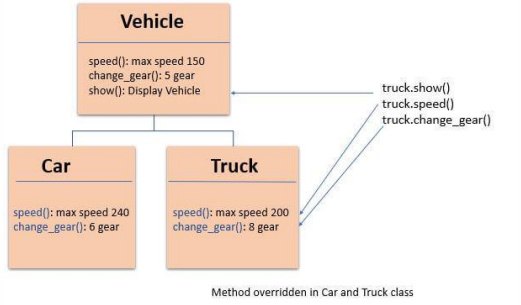
```
max speed is 200 Km/Hour
```

- In the above example, we create two classes named Vehicle (Parent class) and Car (Child class)
  - The class Car extends from the class Vehicle so, all properties of the parent class are available in the child class. In addition to that, the child class redefined the method max_speed()

- **Example 2:**

```
class Animal:
    def eat(self):
        print("This animal is eating")
class Rabbit(Animal):
    def eat(self):
        print("This rabbit is eating a carrot!") # So this will override the main method from parent class

rabbit=Rabbit()
rabbit.eat()

#=> 2 methods with same name and ar/par used by different classes
```

- Polymorphism is mainly used with inheritance. In **inheritance**, child class inherits the attributes and methods of a parent class
  - The existing class is called a base class or parent class, and the new class is called a subclass or child class or derived class
- Using **method overriding** polymorphism allows us to define methods in the child class that have the same name as the methods in the parent class
  - This **process of re-implementing the inherited method in the child class** is known as Method Overriding
- **Advantage of method overriding:**
  - It is effective when we want to extend the functionality by altering the inherited method. Or the method inherited from the parent class doesn't fulfill the need of a child class, so we need to re-implement the same method in the child class in a different way
  - Method overriding is useful when a parent class has multiple child classes, and one of that child class wants to redefine the method. The other child classes can use the parent class method. Due to this, we don't need to modification the parent class code
- In polymorphism, **Python first checks the object's class type and executes the appropriate method** when we call the method
  - For example, If you create the Car object, then Python calls the speed() method from a Car class
    - **Example1 of Method Overriding**
      - In this example, we have a vehicle class as a parent and a 'Car' and 'Truck' as its sub-class. But each vehicle can have a different seating capacity, speed, etc., so we can have the same instance method name in each class but with a different implementation. Using this code can be extended and easily maintained over time



Method overridden in Car and Truck class

```
class Vehicle:

    def __init__(self, name, color, price):
        self.name = name
```

```
        self.color = color
        self.price = price

    def show(self):
        print('Details:', self.name, self.color, self.price)

    def max_speed(self):
        print('Vehicle max speed is 150')

    def change_gear(self):
        print('Vehicle change 6 gear')


# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 240')

    def change_gear(self):
        print('Car change 7 gear')


# Car Object
car = Car('Car x1', 'Red', 20000)
car.show()
# calls methods from Car class
car.max_speed()
car.change_gear()

# Vehicle Object
vehicle = Vehicle('Truck x1', 'white', 75000)
vehicle.show()
# calls method from a Vehicle class
vehicle.max_speed()
vehicle.change_gear()
```

Output:

```
Details: Car x1 Red 20000
Car max speed is 240
Car change 7 gear

Details: Truck x1 white 75000
Vehicle max speed is 150
Vehicle change 6 gear
```

- As you can see, due to polymorphism, the Python interpreter recognizes that the max_speed() and change_gear() methods are overridden for the car object
  - So, it uses the one defined in the child class (Car)
  - On the other hand, the show() method isn't overridden in the Car class, so it is used from the Vehicle class

- **Example 2:**

```
class Mercury:
    def weight(self, w):
        print(w * 0.38)


class Mars(Mercury):
    def weight(self, w):
        print(w * 0.38)


class Earth(Mars):
    def weight(self, w):
        print(w)
```

```
obj = Mercury()
obj.weight(50)
obj = Mars()
obj.weight(50)
obj = Earth()
obj.weight(50)
```

```
19.0
19.0
50
```

IV.4. Duck Typing Philosophy of Python:

- It is derived from the following quote:
  "If it looks like a duck and quacks like a duck, it's a duck."
  - It means the Class which defined the methods is not important rather than the method it defines
  - We don't check types at all in this functionality, we check for the methods and their definition instead

- We can understand it with an **Example** similar to method overriding:

```
class Mercury:
    def weight(self, w):
        print(w * 0.38)


class Mars:
    def weight(self, w):
        print(w * 0.38)


class Earth:
    def weight(self, w):
        print(w)


for obj in Mercury(), Earth(), Mars():
    obj.weight(50)
```

```
19.0
50
19.0
```

- In the above example, we can see we have called the same method name in every class but their functionality is different, and that is what we focus on

Differences between method overriding and duck typing:

- **Polymorphism** (in the context of object-oriented programming) means a subclass can override a method of the base class. This means a method of a class can do different things in subclasses
  - For example: a class Animal can have a method talk() and the subclasses Dog and Cat of Animal can let the method talk() make different sounds
  - Method overloading(Compile time Polymorphism)
    - This is an **Example for Polymorphism** in Python:

```
class Animal:
   def __init__(self, name):  # Constructor of the class
      self.name = name

   def talk(self):  # Abstract method, defined by convention only
      raise NotImplementedError("Subclass must implement abstract method")

class Cat(Animal):
   def talk(self):
      return 'Meow!'

class Dog(Animal):
   def talk(self):
      return 'Woof! Woof!'

animals = [Cat('Missy'),
         Cat('Mr. Mistoffelees'),
         Dog('Lassie')]

for animal in animals:
   print(animal)
   print(animal.name + ': ' + animal.talk())
```

- **Duck typing** **means code will simply accept any object that has a particular method**. Let's say we have the following code: animal.quack()
  - **If the given object animal has the method we want to call then we're good** (no additional type requirements needed). It does not matter whether animal is actually a Duck or a different animal which also happens to quack. That's why it is called duck typing: if it looks like a duck (e.g., it has a method called quack() then we can act as if that object is a duck)
  - Method overriding(Run Time Polymorphism)
    - This is an **Example for duck Typing** in Python:

```
class Duck:
   def quack(self):
      print("Quaaaaaack!")

   def feathers(self):
      print("The duck has white and gray feathers.")

   def name(self):
      print("ITS A DUCK NO NAME")

class Person:
   def quack(self):
      print("The person imitates a duck.")

   def feathers(self):
      print("The person takes a feather from the ground and shows it.")

   def name(self):
      print("John Smith")

def in_the_forest(duck):
   duck.quack()
   duck.feathers()
   duck.name()
```

```
def game():
    for element in [Duck(), Person()]:
        in_the_forest(element)

game()
```

- **Another Example:**

```
class PyCharm:
    """ Duck typing is basically passing
    the object through the caller and then you will
    have access to its def's"""
    def execute(self):
        print("compiling")
        print("running")

class Laptop:
    def code(self,ide): # passing the object as parameter, you can give here any name
        ide.execute() # calling the def of the object

ide= PyCharm()

laptop_1 = Laptop()

laptop_1.code(ide) # you pass in the object as argument so you have to create it
```

Differences between duck typing and Ask Forgiveness than Permission (EAFP):

- **Duck Typing: is actually passing the object as an argument and if it has that particular function then it will do what we ask it to do**

```
# Duck typing means that we do not care what type of object we are working with, we only care for our object can do what we ask it to do

class Duck:

    def quack(self):
        print('Quack, quack')

    def fly(self):
        print('Flap', 'Flap!')

class Person:

    def quack(self):
        print("I'm Quaking Like a Duck!")

    def fly(self):
        print("I'm Flapping my Arms!")

# so it actually is passing the object as an argument and if it has that particular function then it will do what we ask it to do
def quack_and_fly(thing):
    thing.quack()
    thing.fly()

    print()

d = Duck()
quack_and_fly(d) # passing the instance to the function

p = Person()
quack_and_fly(p) # passing the instance to the function
```

- So, when people talk about the fact that it is easier to **ask forgiveness than permission** then it means try to do something and if it works then great and if not then just handle that error:

```
# Duck Typing and Easier to ask forgiveness than permission (EAFP)
# Duck typing means that we do not care what type of object we are working with, we only care for our object can do what we ask it to do

class Duck:

    def quack(self):
        print('Quack, quack')
```

```
    def fly(self):
        print('Flap', 'Flap!')

class Person:

    def quack(self):
        print("I'm Quaking Like a Duck!")

    def fly(self):
        print("I'm Flapping my Arms!")

# def quack_and_fly(thing):
#     # LBYL (Non-Pythonic)
#     if hasattr(thing, 'quack'): # so here we check first if the attribute exists = does this 'thing' has attribute of quack
#         if callable(thing.quack): # second we check if it does have attribute thing of duck to see if it's callable and if it is callable then we finally run it
#             thing.quack()
#     if hasattr(thing, 'fly'): # same as above
#         if callable(thing.fly):
#             thing.fly()
# so instead of the above which is complicated to ask permission every step of the way, we will use:
def quack_and_fly(thing):
    # EAFP (Pythonic)
    try: # so we are saying hey, try to run these methods and if you have an attribute error then print that error
        thing.quack()
        thing.fly()
    except AttributeError as e:
        print(e)

d = Duck()
quack_and_fly(d) # passing the instance to the function

p = Person()
quack_and_fly(p) # passing the instance to the function
```

- **Same as Above can be used on Dictionaries, below is LBYL (can I do this, and this and this….) vs EAFP:**

```
# person = {'name': 'Jesse','age':23,'job':'Programmer}
person = {'name': 'Jesse','age':30}

#LBYL (Non-Pythonic): <- LBYL = "look before you leap"  = is the traditional programming style in which we check if a piece of code is going to work before actually running it
if 'name' in person and 'age' in person and 'job' in person:
    print("I'm {name}. I'm {age} years old and I am a {job}".format(**person))
else:
    print("Missing some keys")

#EAFP (Pythonic): <- EAFP = "Easier to Ask for Forgiveness than Permission" = Python community prefers Easier to Ask for Forgiveness than Permission. Because let's say that your code has many conditions that could go wrong. If you're using Look Before You Leap, then you'll have a lot of if-else statements for checking edge cases
try:
    print("I'm {name}. I'm {age} years old and I am a {job}".format(**person))
except KeyError as e:
    print("Missing {} key".format(e))
```

  - You can see that instead of asking permission lots of times we use EAFP to be easier

- Same concept can be used also **on lists**:

```
my_list = [1, 2, 3, 4, 5, 6]
# non-Pythonic = below is asking permission which is non pythonic
if len(my_list) >= 6:
    print(my_list[5])
else:
    print('That index does not exist')
# Pythonic
# so wew are not asking permission to do something
# we will try to do it and if we can't e will handle it the way we want to handle it
try:
    print(my_list[5])
except IndexError:
    print('That index does not exist')
```

C.6.2.4. Tools used by OOP:

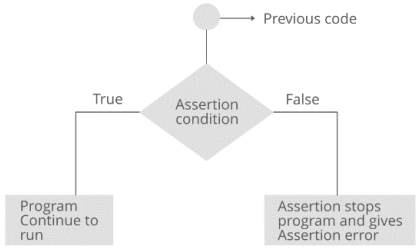
A ▸ Python assert keyword:

- **Assertions** in any programming language are the debugging tools that help in the smooth flow of code
- Assertions are mainly assumptions that a programmer knows or always wants to be true and hence puts them in code, so that failure of these doesn't allow the code to execute further
- In simpler terms, we can say that assertion is the Boolean expression that checks if the statement is True or False

-> If the statement is true then it does nothing and continues the execution, but if the statement is False then it stops the execution of the program and throws an error

-> Let us look at the flowchart of the assertion

Flowchart of Assertion



▸ a) assert Keyword in Python:

- In python, **assert** keyword helps in achieving this task.
- This statement takes as input a Boolean condition, which when returns true doesn't do anything and continues the normal flow of execution

-> but if it is computed to be false, then it raises an AssertionError along with the optional message provided

- ***Syntax :*** *assert condition, error_message(optional)*
- ***Parameters :***
- ***condition :*** *The boolean condition returning true or false.*
- ***error_message :*** *The optional argument to be printed in console in case of AssertionError*
- ***Returns:***

▸ *Returns AssertionError, in case the condition evaluates to false along with the error message which when provided.*

▸ **Example 1**: Python assert keyword without error message

```
# Python 3 code to demonstrate
# working of assert

# initializing number
a = 4
b = 0
# using assert to check for 0
print("The value of a / b is : ")
assert b != 0
print(a / b)
```

Output :

AssertionError:

Example 2: Python assert keyword with error message

```
# Python 3 code to demonstrate
# working of assert
# initializing number
a = 4
b = 0

# using assert to check for 0
```

```
print("The value of a / b is : ")
assert b != 0, "Zero Division Error"
print(a / b)
```

Output:
AssertionError: Zero Division Error

b) Practical Application:

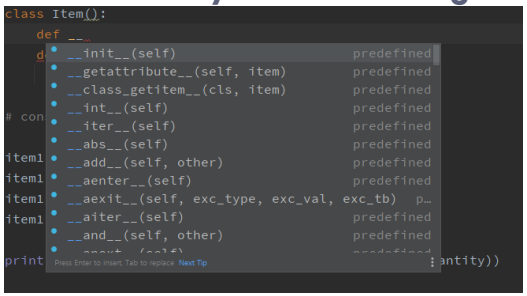
- This has a much greater utility in Testing and Quality Assurance role in any development domain
- Different types of assertions are used depending upon the application
- Below is the simpler demonstration of a program that only allows only the batch with all hot food to be dispatched, else rejects the whole batch

```
# Python 3 code to demonstrate
# working of assert
# Application

# initializing list of foods temperatures
batch = [ 40, 26, 39, 30, 25, 21]

# initializing cut temperature
cut = 26

# using assert to check for temperature greater than cut
for i in batch:
   assert i >= 26, "Batch is Rejected"
   print (str(i) + " is O.K" )
```

Output :
40 is O.K
26 is O.K
39 is O.K
30 is O.K
Runtime Exception :
AssertionError: Batch is Rejected

B ▸ Magic (Dunder, Special) Methods:

• Make your code more Pythonic with Magic Methods •



- **What are magic methods?**
  - **They're everything in object-oriented Python. They're special methods that you can define to add "magic" to your classes**
- They're always surrounded by double underscores (e.g. __init__ or __lt__)
- They're also not as well documented as they need to be. All of the magic methods for Python appear in the same section in the Python docs, but they're scattered about and only loosely organized
  - There's hardly an example to be found in that section (and that may very well be by design, since they're all detailed in the *language reference*, along with boring syntax descriptions, etc.)
- Dunder means "Double Under (Underscores)". These are commonly used for operator overloading
  - Few examples for magic methods are: __init__, __add__, __len__, __repr__,__lt__ is a method that is useful to define the less-than operator. __gt__ is useful to define greater than, etc.
- Magical methods are something useful to define operators in python
- These magical methods cannot be called directly, invocation happens internally from the class on a certain action. These special methods are useful in operator overloading
- Thus, you can use the appropriate magic methods to add various functionalities in your **custom class**
- For example of internal invocation, when you add two numbers using the + operator, internally, the __add__() method will be called
  - Consider the following example:

```
num = 10
num += 5
print(num) # 15

# we can write the above as:
num = 10
print(num.__add__(5))
```

```
from operator import __add__

num = 10
num += 5
print(num) # 15

# we can write the above as:
num = 10

# we can use import operator import __add__ to write the magic method in its direct form
__add__(num, 10)
```

- As you can see, when you do num+= 5, the + operator calls the __add__(5) method. You can also call num.__add__(5) directly which will give the same result
  - However, as mentioned before, magic methods are not meant to be called directly, but internally, through some other methods o r actions
- So, whenever we create an object, Python invokes __init__ method to instantiate the object and whenever we use an operator, Python internally calls the corresponding magic method

```
class Fruit:
  def __init__(self):
     print("Object Created")
obj1 = Fruit()
```

Output

```
Object Created
```

- We can see in the above code example, we never directly called the __init__ method. Python invoked it as soon as we defined an object
  - In order to make the overloaded behavior available in your own custom class, the corresponding magic method should be overridden
    - For example, in order to use the + operator with objects of a user-defined class, it should include the __add__() method
      - In following example, a class named distance is defined with two instance attributes - ft and inch. The addition of these two distance objects is desired to be performed using the overloading + operator
        - To achieve this, the magic method __add__() is overridden, which performs the addition of the ft and inch attributes of the two objects. The __str__() method returns the object's string representation

```
class distance:
   def __init__(self, x=None,y=None):
      self.ft=x
      self.inch=y
   def __add__(self,x):
      temp=distance()
      temp.ft=self.ft+x.ft
      temp.inch=self.inch+x.inch
      if temp.inch>=12:
         temp.ft+=1
         temp.inch-=12
         return temp
   def __str__(self):
      return 'ft:'+str(self.ft)+' in: '+str(self.inch)

d1=distance(3,10)
d2=distance(4,4)
print("d1= {} d2={}".format(d1, d2)) # d1= ft:3 in: 10 d2=ft:4 in: 4

d3=d1+d2
print(d3) # ft:8 in: 2
```

- Built-in classes in Python define many magic methods. Use the dir() function to see the number of magic methods inherited by a class
  - For example, the following lists all the attributes and methods defined in the int class

```
print(dir(int))
```

output:

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__',
'__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__',
'__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__',
'__xor__', 'as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

  - As you can see above, the int class includes various magic methods surrounded by double underscores
- Magic methods are most frequently used to define overloaded behaviors of predefined operators in Python. For instance, arithmetic operators by default operate upon numeric operands
  - This means that numeric objects must be used along with operators like +, -, *, /, etc. The + operator is also defined as a concatenation operator in string, list and tuple classes
    - We can say that the + operator is overloaded
- Some of the magic methods in Python directly map to built-in functions; in this case, how to invoke them is fairly obvious. However, in other cases, the invocation is far less obvious
  - This appendix is devoted to exposing non-obvious syntax that leads to magic methods getting called

Magic Method	When it gets invoked (example)	Explanation
__new__(cls [,...])	instance = MyClass(arg1, arg2)	__new__ is called on instance creation
__init__(self [,...])	instance = MyClass(arg1, arg2)	__init__ is called on instance creation
__cmp__(self, other)	self == other, self > other, etc.	Called for any comparison
__pos__(self)	+self	Unary plus sign
__neg__(self)	-self	Unary minus sign

Magic Method	When it gets invoked (example)	Explanation
__invert__(self)	~self	Bitwise inversion
__index__(self)	x[self]	Conversion when object is used as index
__nonzero__(self)	bool(self)	Boolean value of the object
__getattr__(self, name)	self.name # name doesn't exist	Accessing nonexistent attribute
__setattr__(self, name, val)	self.name = val	Assigning to an attribute
__delattr__(self, name)	del self.name	Deleting an attribute
__getattribute__(self, name)	self.name	Accessing any attribute
__getitem__(self, key)	self[key]	Accessing an item using an index
__setitem__(self, key, val)	self[key] = val	Assigning to an item using an index
__delitem__(self, key)	del self[key]	Deleting an item using an index
__iter__(self)	for x in self	Iteration
__contains__(self, value)	value in self, value not in self	Membership tests using in
__call__(self [,...])	self(args)	"Calling" an instance
__enter__(self)	with self as x:	with statement context managers
__exit__(self, exc, val, trace)	with self as x:	with statement context managers
__getstate__(self)	pickle.dump(pkl_file, self)	Pickling
__setstate__(self)	data = pickle.load(pkl_file)	Pickling

• How to Call Magic Methods •

Initialization and Construction	Description
__new__(cls, other)	To get called in an object's instantiation.
__init__(self, other)	To get called by the __new__ method.
__del__(self)	Destructor method.
Unary operators and functions	**Description**
__pos__(self)	To get called for unary positive e.g. +someobject.
__neg__(self)	To get called for unary negative e.g. -someobject.
__abs__(self)	To get called by built-in abs() function.
__invert__(self)	To get called for inversion using the ~ operator.
__round__(self,n)	To get called by built-in round() function.
__floor__(self)	To get called by built-in math.floor() function.
__ceil__(self)	To get called by built-in math.ceil() function.
__trunc__(self)	To get called by built-in math.trunc() function.
Augmented Assignment	**Description**
__iadd__(self, other)	To get called on addition with assignment e.g. a +=b.
__isub__(self, other)	To get called on subtraction with assignment e.g. a -=b.
__imul__(self, other)	To get called on multiplication with assignment e.g. a *=b.

Augmented Assignment	Description
__ifloordiv__(self, other)	To get called on integer division with assignment e.g. a //=b.
__idiv__(self, other)	To get called on division with assignment e.g. a /=b.
__itruediv__(self, other)	To get called on true division with assignment
__imod__(self, other)	To get called on modulo with assignment e.g. a%=b.
__ipow__(self, other)	To get called on exponentswith assignment e.g. a **=b.
__ilshift__(self, other)	To get called on left bitwise shift with assignment e.g. a<<=b.
__irshift__(self, other)	To get called on right bitwise shift with assignment e.g. a >>=b.
__iand__(self, other)	To get called on bitwise AND with assignment e.g. a&=b.
__ior__(self, other)	To get called on bitwise OR with assignment e.g. a|=b.
__ixor__(self, other)	To get called on bitwise XOR with assignment e.g. a ^=b.
Type Conversion Magic Methods	**Description**
__int__(self)	To get called by built-int int() method to convert a type to an int.
__float__(self)	To get called by built-int float() method to convert a type to float.
__complex__(self)	To get called by built-int complex() method to convert a type to complex.
__oct__(self)	To get called by built-int oct() method to convert a type to octal.
__hex__(self)	To get called by built-int hex() method to convert a type to hexadecimal.
__index__(self)	To get called on type conversion to an int when the object is used in a slice expression.
__trunc__(self)	To get called from math.trunc() method.
String Magic Methods	**Description**
__str__(self)	To get called by built-int str() method to return a string representation of a type.
__repr__(self)	To get called by built-int repr() method to return a machine readable representation of a type.
__unicode__(self)	To get called by built-int unicode() method to return an unicode string of a type.
__format__(self, formatstr)	To get called by built-int string.format() method to return a new style of string.
__hash__(self)	To get called by built-int hash() method to return an integer.
__nonzero__(self)	To get called by built-int bool() method to return True or False.
__dir__(self)	To get called by built-int dir() method to return a list of attributes of a class.
__sizeof__(self)	To get called by built-int sys.getsizeof() method to return the size of an object.
Attribute Magic Methods	**Description**
__getattr__(self, name)	Is called when the accessing attribute of a class that does not exist.
__setattr__(self, name, value)	Is called when assigning a value to the attribute of a class.
__delattr__(self, name)	Is called when deleting an attribute of a class.
Operator Magic Methods	**Description**
__add__(self, other)	To get called on add operation using + operator
__sub__(self, other)	To get called on subtraction operation using - operator.
__mul__(self, other)	To get called on multiplication operation using * operator.
__floordiv__(self, other)	To get called on floor division operation using // operator.
__truediv__(self, other)	To get called on division operation using / operator.
__mod__(self, other)	To get called on modulo operation using % operator.
__pow__(self, other[, modulo])	To get called on calculating the power using ** operator.
__lt__(self, other)	To get called on comparison using < operator.
__le__(self, other)	To get called on comparison using <= operator.
__eq__(self, other)	To get called on comparison using == operator.
__ne__(self, other)	To get called on comparison using != operator.
__ge__(self, other)	To get called on comparison using >= operator.

a) Construction and Initialization:

- Everyone knows the most basic magic method, `__init__`. It's the way that we can define the initialization behavior of an object
  - However, when we call `x = SomeClass()`, `__init__` is not the first thing to get called
  - Actually, it's a method called **`__new__`**, which actually **creates the instance**, then passes any arguments at creation on to the initializer
  - At the other end of the object's lifespan, there's `__del__`. Let's take a closer look at these 3 magic methods:

  `__new__(cls, [...)`

  `__new__` **is the first method to get called in an object's instantiation. It takes the class, then any other arguments that it will pass along to** `__init__`

  `__new__` is used fairly rarely, but it does have its **purposes**, particularly when **subclassing an immutable type like a tuple or a string**

__new__ because it's not too useful
__init__(self, [...]):
The initializer for the class. It gets passed whatever the primary constructor was called with (so, for example, if we called x = SomeClass(10, 'foo'), __init__ would get passed 10 and 'foo' as arguments. __init__ is almost universally used in Python class definitions
__del__(self):
If __new__ and __init__ formed the constructor of the object, __del__ is the destructor
It doesn't implement behavior for the statement del x (so that code would not translate to x.__del__()). Rather, it defines behavior for when an object is garbage collected.
It can be quite useful for objects that might require extra cleanup upon deletion, like sockets or file objects
Be careful, however, as there is no guarantee that __del__ will be executed if the object is still alive when the interpreter exits, so __del__ can't serve as a replacement for good coding practices (like always closing a connection when you're done with it. In fact, __del__ should almost never be used because of the precarious circumstances under which it is called; use it with caution!

- Putting it all together, here's an example of __init__ and __del__ in action:

```
from os.path import join

class FileObject:
  '''Wrapper for file objects to make sure the file gets closed on deletion.'''

  def __init__(self, filepath='~', filename='sample.txt'):
    # open a file filename in filepath in read and write mode
    self.file = open(join(filepath, filename), 'r+')

  def __del__(self):
    self.file.close()
    del self.file
```

__new__() method:

- Languages such as Java and C# use the new operator to create a new instance of a class. In Python the __new__() magic method is implicitly called before the __init__() method
- The __new__() method returns a new object, which is then initialized by __init__()
- When we talk about magic method __new__ we also need to talk about __init__
- These methods will be called when you instantiate(The process of creating instance from class is called instantiation). That is when you create instance
  - The magic method __new__ will be called when instance is being created. Using this method you can customize the instance creation
  - This is only the method which will be called first then __init__ will be called to initialize instance when you are creating instance
  - Method __new__ will take class reference as the first argument followed by arguments which are passed to constructor(Arguments passed to call of class to create instance)
- Method __new__ is responsible to create instance, so you can use this method to customize object creation. Typically method __new__ will return the created instance object reference
  - Method __init__ will be called once __new__ method completed execution

```
class Animal:
  def __new__(cls, legs):
    if legs == 2:
      return Biped()
    else:
      return Quadruped()

class Biped:
  def __init__(self):
    print("Initializing 2-legged animal")

class Quadruped:
  def __init__(self):
    print("Initializing 4-legged animal")

anim1 = Animal(legs=4)
anim1 = Animal(legs=2)
```

You can use __new__ to conditionally create an instance from a class

Key Takeaways

- In most cases, you don't need
- __new__ is called before __init__.
- __new__ returns a instance of class.
- __init__ receives the instances of the class returned by __new__.
- Use __init__ to initilize value.

__init__() Constructor (Initializer) method:

- **part of magic methods**
- **use constructor to avoid hard-coding of the attributes for the instances created**
- It is an instance method for initializing new Objects with values for properties. We define it using the keyword `__init__()`
- using the `__init__()` method we can implement constructor to initialize the object
- **Constructor** is a special method used to create and initialize an object of a class. On the other hand, a destructor is used to destroy the object
  - The constructor is executed automatically at the time of object creation
  - The primary use of a constructor is to declare and initialize data member/ instance variables of a class. The constructor contains a collection of statements (i.e., instructions) that executes at the time of object creation to initialize the attributes of an object
    For example, when we execute `obj = Sample()`, Python gets to know that obj is an object of class `Sample` and calls the constructor of that class to create an object

Note:
In Python, internally, the `__new__` is the method that creates the object, and `__del__` method is called to destroy the object when the reference count for that object becomes zero

- In Python, Object creation is divided into two parts in **Object Creation** and **Object initialization**
  - Internally, the `__new__` is the method that **creates the object**
  - And, using the `__init__()` method we can **implement constructor to initialize the object**
    Syntax of a constructor:

```
def __init__(self):
    # body of the constructor
```

Where,

- `def`: The keyword is used to define function
- `__init__()` Method: It is a reserved method. This method gets called as soon as an object of a class is instantiated
- `self`: The first argument `self` refers to the current object. It binds the instance to the `__init__()` method. It's usually named `self` to follow the naming convention

Note: The `__init__()` method arguments are optional. We can define a constructor with any number of arguments

In the example below we add another parameter 'name':

```
class Item():
    def __init__(self, name):
        print(f"an instance created: {name}")
    def calculate_total_price(self, x, y):
        return x * y

item1 = Item("Phone")
item1.name = 'Phone'
item1.price = 100
item1.quantity = 5

item2 = Item("Laptop")
item2.name = 'Laptop'
item2.price = 1000
item2.quantity = 3
```

- As __init__ method receives 'self' as parameter, we can also dynamically assign other attributes to this instance using __init__ magic method
  (still working with instance attributes below):

```
# dynamic attribute assignment thanks to self.name = name, same goes for price and quantity
# always take care of the attributes you want to assign to an object inside the constructor
```

```
class Item():
    def __init__(self, name: str, price: float, quantity=0):# you set default parameter to 0 as you do not know exact number, so no need to pass a parameter if you do it this way
        # run validations to the received arguments
        # if you never want to receive a negative number of quantity of price, by the typing above(price: float) it cannot be done
        # you can use assert statements for that
        # assert statement is a statement keyword that is used to check if there is a match between what is happening with your expectations
        # using the assert statement it will allow us to validate the arguments that we received and allow us to catch up the bugs asap
        assert price >= 0, f"Price {price} is not greater or equal to zero"
        assert quantity >= 0, f"Price {quantity} is not greater or equal to zero"

        # assign to self object
        self.name = name #self.name will store the value
        self.price = price
        self.quantity = quantity

    def calculate_total_price(self):
        return self.price * self.quantity


item1 = Item("Phone" , 100, 1)
item2 = Item("Laptop", 1000, 3)

print(item1.name)
print(item2.name)
print(item1.price)
print(item2.price)
print(item1.quantity)
print(item2.quantity)

# you can assign attributes to specific instances individually
# ex. if you want to find out if the laptop has a numpad or not and this is not something to be assigned to a phone
item2.has_numpad = False

print(item1.calculate_total_price())
print(item2.calculate_total_price())
print(item2.has_numpad)
```

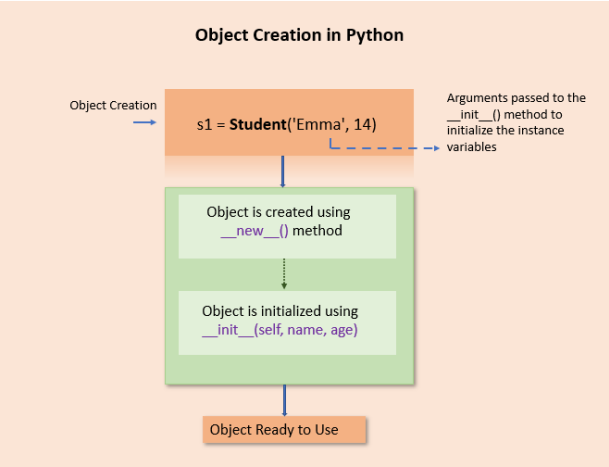
- In below another example, we'll create a Class **Student** with an instance variable student name
  - we'll see how to use a constructor to initialize the student name at the time of object creation

```
class Student:

    # constructor
    # initialize instance variable
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('All variables initialized')

    # instance Method
    def show(self):
        print('Hello, my name is', self.name)


# create object using constructor
s1 = Student('Emma')
s1.show()
```

Output

```
Inside Constructor
All variables initialized
Hello, my name is Emma
```

- In the above example, an object s1 is created using the constructor
- While creating a Student object name is passed as an argument to the __init__() method to initialize the object
- Similarly, various objects of the Student class can be created by passing different names as arguments



Create an object in Python using a constructor

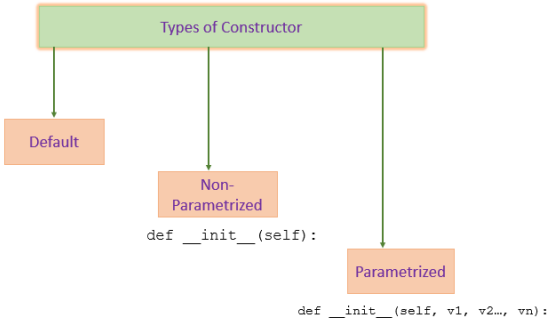
Note:

- For every object, the constructor will be executed only once. For example, if we create four objects, the constructor is called four times
- In Python, every class has a constructor, but it's not required to define it explicitly. Defining constructors in class is optional
- Python will provide a default constructor if no constructor is defined

I. Types of Constructors:

In Python, we have the following three types of constructors:

a) Default Constructor
b) Non-parametrized constructor
c) Parameterized constructor



a) Default Constructor:

- Python will provide a default constructor if no constructor is defined. Python adds a default constructor when we do not include the constructor in the class or forget to declare it. It does not perform any task but initializes the objects. It is an empty constructor without a body
- If you do not implement any constructor in your class or forget to declare it, the Python inserts a default constructor into your code on your behalf, This constructor is known as the default constructor
- It does not perform any task but initializes the objects. It is an empty constructor without a body

Note:

- The default constructor is not present in the source py file. It is inserted into the code during compilation if not exists
- If you implement your constructor, then the default constructor will not be added

Example:

```
class Employee:

    def display(self):
        print('Inside Display')

emp = Employee()
emp.display()
```

Output

```
Inside Display
```

- As you can see in the above example, we do not have a constructor, but we can still create an object for the class because Python added the default constructor during a program compilation

b) Non-Parametrized Constructor:

- A constructor without any arguments is called a non-parameterized constructor. This type of constructor is used to initialize each object with default values
- This constructor doesn't accept the arguments during object creation. Instead, it initializes every object with the same set of values

```
class Company:

    # no-argument constructor
    def __init__(self):
        self.name = "PYnative"
        self.address = "ABC Street"

    # a method for printing data members
    def show(self):
        print('Name:', self.name, 'Address:', self.address)

# creating object of the class
cmp = Company()

# calling the instance method using the object
cmp.show()
```

Output

```
Name: PYnative Address: ABC Street
```

- As you can see in the above example, we do not send any argument to a constructor while creating an object

c) Parameterized Constructor:

- A constructor with defined parameters or arguments is called a parameterized constructor. We can pass different values to each object at the time of creation using a parameterized constructor
- The first parameter to constructor is self that is a reference to the being constructed, and the rest of the arguments are provided by the programmer. A parameterized constructor can have any number of arguments
- For example, consider a company that contains thousands of employees. In this case, while creating each employee object, we need to pass a different name, age, and salary. In such cases, use the parameterized constructor

Example:

```
class Employee:
    # parameterized constructor
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

    # display object
```

```
    def show(self):
        print(self.name, self.age, self.salary)

# creating object of the Employee class
emma = Employee('Emma', 23, 7500)
emma.show()

kelly = Employee('Kelly', 25, 8500)
kelly.show()
```

Output

```
Emma 23 7500
Kelly 25 8500
```

- In the above example, we define a parameterized constructor which takes three parameters

II. Constructor with Default Values:

- Python allows us to define a constructor with default values. The default value will be used if we do not pass arguments to the constructor at the time of object creation
- The following example shows how to use the default values with the constructor

Example:

```
class Student:
    # constructor with default values age and classroom
    def __init__(self, name, age=12, classroom=7):
        self.name = name
        self.age = age
        self.classroom = classroom

    # display Student
    def show(self):
        print(self.name, self.age, self.classroom)

# creating object of the Student class
emma = Student('Emma')
emma.show()

kelly = Student('Kelly', 13)
kelly.show()
```

Output

```
Emma 12 7
Kelly 13 7
```

- As you can see, we didn't pass the age and classroom value at the time of object creation, so default values are used

III. Self Keyword in Python:

- As you all know, the class contains instance variables and methods. Whenever we define instance methods for a class, we use self as the first parameter
- Using self, we can **access the instance variable and instance method** of the object
- So, **The first argument self refers to the current object**
- Whenever we call an instance method through an object, the Python compiler implicitly passes object reference as the first argument commonly known as self
- It is not mandatory to name the first parameter as a self. We can give any name whatever we like, but it has to be the first parameter of an instance method

Example:

```
class Student:
    # constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    # self points to the current object
    def show(self):
        # access instance variable using self
        print(self.name, self.age)

# creating first object
emma = Student('Emma', 12)
emma.show()

# creating Second object
kelly = Student('Kelly', 13)
kelly.show()
```

Output

```
Emma 12
Kelly 13
```

VI. Constructor Overloading:

- Constructor overloading is a concept of having more than one constructor with a different parameters list in such a way so that each constructor can perform different tasks
- For example, we can create a three constructor which accepts a different set of parameters
- **Python does not support constructor overloading**. If we define multiple constructors then, the interpreter will consider only the last constructor and throws an error if the sequence of the arguments doesn't match as per the last constructor
- The following example shows the same:

Example:

```
class Student:
    # one argument constructor
    def __init__(self, name):
        print("One arguments constructor")
        self.name = name

    # two argument constructor
    def __init__(self, name, age):
        print("Two arguments constructor")
        self.name = name
        self.age = age

# creating first object
emma = Student('Emma')

# creating Second object
kelly = Student('Kelly', 13)
```

Output

```
TypeError: __init__() missing 1 required positional argument: 'age'
```

- As you can see in the above example, we defined multiple constructors with different arguments
- At the time of object creation, the interpreter executed the second constructor because Python always considers the last constructor
- Internally, the object of the class will always call the last constructor, even if the class has multiple constructors
- In the example when we called a constructor only with one argument, we got a type error

V. Constructor Chaining:

- Constructors are used for instantiating an object. The task of the constructor is to assign value to data members when an object of the class is created
- Constructor chaining is the process of calling one constructor from another constructor from different classes (Parent -> child)
- Constructor chaining is useful when you want to invoke multiple constructors, one after another, by initializing only one instance
- In Python, **constructor chaining is convenient when we are dealing with inheritance**

  -> When an instance of a child class is initialized, the constructors of all the parent classes are first invoked automatically and then, in the end,

the constructor of the child class is invoked

- Using the `super()` method we can invoke the parent class constructor from a child class

Example:

```
class Vehicle:
    # Constructor of Vehicle
    def __init__(self, engine):
        print('Inside Vehicle Constructor')
        self.engine = engine

class Car(Vehicle):
    # Constructor of Car
    def __init__(self, engine, max_speed):
        super().__init__(engine)
        print('Inside Car Constructor')
        self.max_speed = max_speed

class Electric_Car(Car):
    # Constructor of Electric Car
    def __init__(self, engine, max_speed, km_range):
        super().__init__(engine, max_speed)
        print('Inside Electric Car Constructor')
        self.km_range = km_range

# Object of electric car
ev = Electric_Car('1500cc', 240, 750)
print(f'Engine={ev.engine}, Max Speed={ev.max_speed}, Km range={ev.km_range}')
```

Output

```
Inside Vehicle Constructor
Inside Car Constructor
Inside Electric Car Constructor

Engine=1500cc, Max Speed=240, Km range=750
```

VI. Counting the Number of objects of a Class:

- The constructor executes when we create the object of the class. **For every object, the constructor is called only once**
- So, for counting the number of objects of a class, we can add a counter in the constructor, which **increments** by one **after each object creation**

Example:

```
class Employee:
    count = 0
    def __init__(self):
        Employee.count = Employee.count + 1


# creating objects
e1 = Employee()
e2 = Employee()
e2 = Employee()
print("The number of Employee:", Employee.count)
```

Output

```
The number of employee: 3
```

VII. Constructor Return Value:

- In Python, the constructor does not return any value. Therefore, while declaring a constructor, we don't have anything like return type
  -> Instead, a constructor is implicitly called at the time of object instantiation. Thus, it has the sole purpose of initializing the instance variables

- The `__init__()` is required to return None. We can not return something else. If we try to return a non-None value from the `__init__()` method, it will raise TypeError => you do not use return with __init__

Example:

```
class Test:

  def __init__(self, i):
    self.id = i
    return True

d = Test(10)
```

Output

```
TypeError: __init__() should return None, not 'bool'
```

VIII. Conclusion and Quick recap:

- So with constructors we use them in object-oriented programming to design classes and create objects
- The below list contains the summary of the concepts related to constructors:
  - **A constructor is a unique method used to initialize an object of the class**
  - **Python will provide a default constructor if no constructor is defined**
  - Constructor is not a method and **doesn't return anything**. it returns None
  - **In Python, we have three types of constructor default, Non-parametrized, and parameterized constructor**
  - **Using self, we can access the instance variable and instance method of the object. The first argument self refers to the current object**
  - **Constructor overloading is not possible in Python**
  - If the parent class doesn't have a default constructor, then the compiler would not insert a default constructor in the child class
  - **A child class constructor can also invoke the parent class constructor using the `super()` method**

b) Making Operators Work on Custom Classes:

- One of the biggest advantages of using Python's magic methods is that they provide a simple way to make objects behave like built-in types
  - That means you can avoid ugly, counter-intuitive, and nonstandard ways of performing basic operators. In some languages, it's common to do something like this:

```
if instance.equals(other_instance):
  # do something
```

  - You could certainly do this in Python, too, but this adds confusion and is unnecessarily verbose
    - Different libraries might use different names for the same operations, making the client do way more work than necessary
      - With the power of magic methods, however, we can define one method (`__eq__`, in this case), and say what we *mean* instead:

```
if instance == other_instance:
  #do something
```

- That's part of the power of magic methods. The vast majority of them allow us to define meaning for operators so that we can use them on our own classes just like they were built in types

c) Comparison magic methods:

- Python has a whole slew of magic methods designed to implement intuitive comparisons between objects using operators, not awkward method calls
- They also provide a way to override the default Python behavior for comparisons of objects (by reference)
- Here's the list of those methods and what they do:

`__cmp__(self, other)`:
`__cmp__` is the most basic of the comparison magic methods. It actually implements behavior for all of the comparison operators (<, ==, !=, etc.), but it might not do it the way you want (for example, if whether one instance was equal to another were determined by one criterion and whether an instance is greater than another were determined by something else). `__cmp__` should return a negative integer if self < other, zero if self == other, and positive if self > other. It's usually best to define each comparison you need rather than define them all at once, but `__cmp__` can be a good way to save repetition and improve clarity when you need all comparisons implemented with similar criteria

`__eq__(self, other)`:
Defines behavior for the equality operator, ==

`__ne__(self, other)`:
Defines behavior for the inequality operator, !=

`__lt__(self, other)`:
Defines behavior for the less-than operator, <

`__gt__(self, other)`:
Defines behavior for the greater-than operator, `>`
`__le__(self, other)`:
Defines behavior for the less-than-or-equal-to operator, `<=`
`__ge__(self, other)`:
Defines behavior for the greater-than-or-equal-to operator, `>=`

- **For an example**, consider a class to model a word. We might want to compare words lexicographically (by the alphabet), which is the default comparison behavior for strings, but we also might want to do it based on some other criterion, like length or number of syllables. In this example, we'll compare by length. Here's an implementation:

```
class Word(str):
  '''Class for words, defining comparison based on word length.'''

  def __new__(cls, word):
    # Note that we have to use __new__. This is because str is an immutable
    # type, so we have to initialize it early (at creation)
    if ' ' in word:
      print "Value contains spaces. Truncating to first space."
      word = word[:word.index(' ')] # Word is now all chars before first space
    return str.__new__(cls, word)

  def __gt__(self, other):
    return len(self) > len(other)
  def __lt__(self, other):
    return len(self) < len(other)
  def __ge__(self, other):
    return len(self) >= len(other)
  def __le__(self, other):
    return len(self) <= len(other)
```

- Now, we can create two Words (by using Word('foo') and Word('bar')) and compare them based on length. Note, however, that we didn't define `__eq__` and `__ne__`
  - This is because this would lead to some weird behavior (notably that Word('foo') == Word('bar') would evaluate to true)
    - It wouldn't make sense to test for equality based on length, so we fall back on str's implementation of equality
- Now would be a good time to note that you don't have to define every comparison magic method to get rich comparison
  - The standard library has kindly provided us with a class decorator in the module functools that will define all rich comparison methods if you only define `__eq__` and one other (e.g. `__gt__`, `__lt__`, etc.)
    - This feature is only available in Python 2.7, but when you get a chance it saves a great deal of time and effort. You can use it by placing @total_ordering above your class definition

`__ge__()` method:

- The following method is added in the distance class to overload the `>=` operator

```
class distance:
    def __init__(self, x=None,y=None):
        self.ft=x
        self.inch=y
    def __ge__(self, x):
        val1=self.ft*12+self.inch
        val2=x.ft*12+x.inch
        if val1>=val2:
            return True
        else:
            return False
d1=distance(2,1)
d2=distance(4,10)
print(d1>=d2) # False
```

  - This method gets invoked when the `>=` operator is used and returns True or False. Accordingly, the appropriate message can be displayed

`__lt__()` method:

- `__lt__` is a special method for **less than operator(<)**
- Usually, we are using less than operator to check the less than numbers
- This method is not called directly like the less-than operator
- `__lt__` is useful for sorting
- It is useful for operator overloading
- `__lt__` is useful for comparing
- Definition of Python `__lt__` magical operator:
  - As we already know `__lt__` operator cannot be used directly. We can define it as follows:
    `__lt__(self, other)`

Example1:

Using Python __lt__ to compare the weight of two persons:

```
class weight:
  def __init__(self, weight):
    self. weight= weight
  def __lt__(self,other):
    return self.weight<other.weight
a=weight(50)
b=weight(60)
print(a<b)
```

Output

True

- Creating a class named **"weight"**. Defining a **__init__ function**. Inside a function declaring the weight. Creating another function named **__lt__**
  - This function is to compare the weights. Variables a and b holds the weight of two different persons. It compares the weight if the condition is true. It will display the result as True
    - Otherwise False

Example2:

Using Python __lt__ to compare the weight of multiple persons:

```
class weight:
  def __init__(self, weight):
    self. weight= weight
  def __lt__(self,other):
    return self.weight<other.weight
  def __gt__(self,other):
    return self.weight>other.weight
a=weight(50)
b=weight(60)
c=weight(70)
print(a<b and b>c)
```

Output

False

- Creating a class named **"weight"**. Defining a __init__ function. Inside a function declaring the weight. Creating another function named **__lt__** and followed by **__gt__**
  - These functions are to compare the weights. Variables a, b, and c holds the weight of three different persons
    - It compares the weight if the condition is true; it will display the result as True. Otherwise False

Example3:

Using __lt__ for operator overloading:

```
class overload:
  a=0
  def __init__(self,operator):
    self.a=operator
  def __lt__(self,other):
    print("less than operator overloaded")
    if self.a<other.a:
      return True
    else:
      return False
x=overload(15)
y=overload(9)
print(f'x<y = {x < y}')
```

Output

less than operator overloaded

x<y = False

- Creating a class overload. Defining a function init. Creating another function named **lt**.
  - Using an if-else statement inside a function. If the number on the left-hand side is lesser than the right-hand side, it will display True. Otherwise False

Example4:

Using Python __lt__ to sort:

```
import math
from functools import total_ordering
class point:
    def __init__(self,a,b):
        self.a = a
        self.b = b
    def __repr__(self):
        return f'Point(a={self.a}, b={self.b})'
    @property
    def compare(self):
```

```
        return math.sqrt(self.a * self.a + self.b * self.b)
    def __lt__(self, other):
        return self.compare< other.compare
print(sorted([
    point(10, 5), point(30, 10), point(17, 7), point(2, 0)
]))
```

Output

[Point(a=2, b=0), Point(a=10, b=5), Point(a=17, b=7), Point(a=30, b=10)]

- Importing math module. Creating a class named point. It will sort all the points
  - Here we are importing functools that is to support the less-than operator. Using a formatted function to display the output

d) Numeric magic methods:

- Just like you can create ways for instances of your class to be compared with comparison operators, you can define behavior for numeric operators. Buckle your seat belts, folks...there's a lot of these!
- For organization's sake, I've split the numeric magic methods into **5 categories**: unary operators, normal arithmetic operators, reflected arithmetic operators (more on this later), augmented assignment, and type conversions

I. Unary operators and functions:

- Unary operators and functions only have one operand, e.g. negation, absolute value, etc

`__pos__(self)`:
Implements behavior for unary positive (e.g. `+some_object`)

`__neg__(self)`:
Implements behavior for negation (e.g. `-some_object`)

`__abs__(self)`:
Implements behavior for the built in `abs()` function

`__invert__(self)`:
Implements behavior for inversion using the `~` operator. For an explanation on what this does, see the Wikipedia article on bitwise operations

`__round__(self, n)`:
Implements behavior for the built in `round()` function. `n` is the number of decimal places to round to

`__floor__(self)`:
Implements behavior for `math.floor()`, i.e., rounding down to the nearest integer

`__ceil__(self)`:
Implements behavior for `math.ceil()`, i.e., rounding up to the nearest integer

`__trunc__(self)`:
Implements behavior for `math.trunc()`, i.e., truncating to an integral

II. Normal arithmetic operators:

- Now, we cover the typical binary operators (and a function or two): +, -, * and the like. These are, for the most part, pretty self-explanatory

`__add__(self, other)`:
Implements addition

`__sub__(self, other)`:
Implements subtraction

`__mul__(self, other)`:
Implements multiplication

`__floordiv__(self, other)`:
Implements integer division using the `//` operator

`__div__(self, other)`:
Implements division using the `/` operator

`__truediv__(self, other)`:
Implements *true* division. Note that this only works when `from __future__ import division` is in effect

`__mod__(self, other)`:
Implements modulo using the `%` operator

`__divmod__(self, other)`:
Implements behavior for long division using the `divmod()` built in function

`__pow__`
Implements behavior for exponents using the `**` operator

`__lshift__(self, other)`:
Implements left bitwise shift using the `<<` operator

`__rshift__(self, other)`:
Implements right bitwise shift using the `>>` operator

`__and__(self, other)`:
Implements bitwise and using the `&` operator

`__or__(self, other)`:
Implements bitwise or using the `|` operator

`__xor__(self, other)`:
Implements bitwise xor using the `^` operator

III. Reflected arithmetic operators:

- You know how I said I would get to reflected arithmetic in a bit? Some of you might think it's some big, scary, foreign concept. It's actually quite simple. Here's an example:

```
some_object + other
```

- That was "normal" addition. The reflected equivalent is the same thing, except with the operands switched around:

```
other + some_object
```

- So, all of these magic methods do the same thing as their normal equivalents, except the perform the operation with other as the first operand and self as the second, rather than the other way around
- In most cases, the result of a reflected operation is the same as its normal equivalent, so you may just end up defining `__radd__` as calling `__add__` and so on
- Note that the object on the left hand side of the operator (`other` in the example) must not define (or return `NotImplemented`) for its definition of the non-reflected version of an operation
- For instance, in the example, `some_object.__radd__` will only be called if `other` does not define `__add__`

`__radd__(self, other)`:
Implements reflected addition
`__rsub__(self, other)`:
Implements reflected subtraction
`__rmul__(self, other)`:
Implements reflected multiplication
`__rfloordiv__(self, other)`:
Implements reflected integer division using the `//` operator
`__rdiv__(self, other)`:
Implements reflected division using the `/` operator
`__rtruediv__(self, other)`:
Implements reflected *true* division. Note that this only works when `from __future__ import division` is in effect
`__rmod__(self, other)`:
Implements reflected modulo using the `%` operator.
`__rdivmod__(self, other)`:
Implements behavior for long division using the `divmod()` built in function, when `divmod(other, self)` is called
`__rpow__`
Implements behavior for reflected exponents using the `**` operator
`__rlshift__(self, other)`:
Implements reflected left bitwise shift using the `<<` operator
`__rrshift__(self, other)`:
Implements reflected right bitwise shift using the `>>` operator
`__rand__(self, other)`:
Implements reflected bitwise and using the `&` operator
`__ror__(self, other)`:
Implements reflected bitwise or using the `|` operator
`__rxor__(self, other)`:
Implements reflected bitwise xor using the `^` operator

IV. Augmented assignment:

- Python also has a wide variety of magic methods to allow custom behavior to be defined for augmented assignment
- You're probably already familiar with augmented assignment, it combines "normal" operators with assignment. If you still don't know what I'm talking about, here's an example:

```
x = 5
x += 1 # in other words x = x + 1
```

- Each of these methods should return the value that the variable on the left hand side should be assigned to (for instance, for `a += b`, `__iadd__` might return `a + b`, which would be assigned to `a`). Here's the list:

`__iadd__(self, other)`:
Implements addition with assignment
`__isub__(self, other)`:
Implements subtraction with assignment
`__imul__(self, other)`:
Implements multiplication with assignment
`__ifloordiv__(self, other)`:
Implements integer division with assignment using the `//=` operator
`__idiv__(self, other)`:
Implements division with assignment using the `/=` operator
`__itruediv__(self, other)`:
Implements *true* division with assignment. Note that this only works when `from __future__ import division` is in effect
`__imod__(self, other)`:
Implements modulo with assignment using the `%=` operator
`__ipow__`
Implements behavior for exponents with assignment using the `**=` operator
`__ilshift__(self, other)`:

Implements left bitwise shift with assignment using the <<= operator

__irshift__(self, other):

Implements right bitwise shift with assignment using the >>= operator

__iand__(self, other):

Implements bitwise and with assignment using the &= operator

__ior__(self, other):

Implements bitwise or with assignment using the |= operator

__ixor__(self, other):

Implements bitwise xor with assignment using the ^= operator

V. Type conversion magic methods:

- Python also has an array of magic methods designed to implement behavior for built in type conversion functions like float(). Here they are:

__int__(self):

Implements type conversion to int

__long__(self):

Implements type conversion to long

__float__(self):

Implements type conversion to float

__complex__(self):

Implements type conversion to complex

__oct__(self):

Implements type conversion to octal

__hex__(self):

Implements type conversion to hexadecimal

__index__(self):

Implements type conversion to an int when the object is used in a slice expression. If you define a custom numeric type that might be used in slicing, you should define __index__

__trunc__(self):

Called when math.trunc(self) is called. __trunc__ should return the value of `self truncated to an integral type (usually a long)

__coerce__(self, other)

Method to implement mixed mode arithmetic. __coerce__ should return None if type conversion is impossible Otherwise, it should return a pair (2-tuple) of self and other, manipulated to have the same type

e) Representing your Classes:

- It's often useful to have a string representation of a class
- In Python, there are a few methods that you can implement in your class definition to customize how built in functions that return representations of your class behave

__str__(self):

Defines behavior for when str() is called on an instance of your class

__repr__(self):

Defines behavior for when repr() is called on an instance of your class. The major difference between str() and repr() is intended audience. repr() is intended to produce output that is mostly machine-readable (in many cases, it could be valid Python code even), whereas str() is intended to be human-readable

__unicode__(self):

Defines behavior for when unicode() is called on an instance of your class. unicode() is like str(), but it returns a unicode string. Be wary: if a client calls str() on an instance of your class and you've only defined __unicode__(), it won't work. You should always try to define __str__() as well in case someone doesn't have the luxury of using unicode

__format__(self, formatstr):

Defines behavior for when an instance of your class is used in new-style string formatting. For instance, "Hello, {0:abc}!".format(a) would lead to the call a.__format__("abc"). This can be useful for defining your own numerical or string types that you might like to give special formatting options

__hash__(self):

Defines behavior for when hash() is called on an instance of your class. It has to return an integer, and its result is used for quick key comparison in dictionaries. Note that this usually entails implementing __eq__ as well. Live by the following rule: a == b implies hash(a) == hash(b)

__nonzero__(self):

Defines behavior for when bool() is called on an instance of your class. Should return True or False, depending on whether you would want to consider the instance to be True or False

__dir__(self):

Defines behavior for when dir() is called on an instance of your class. This method should return a list of attributes for the user. Typically, implementing __dir__ is unnecessary, but it can be vitally important for interactive use of your classes if you redefine __getattr__ or __getattribute__ (which you will see in the next section) or are otherwise dynamically generating attributes

__sizeof__(self):

Defines behavior for when sys.getsizeof() is called on an instance of your class. This should return the size of your object, in bytes. This is generally more useful for Python classes implemented in C extensions, but it helps to be aware of it

__str__() method:

- **When you use the print() function to display the instance of the Person class, the print() function shows the memory address of that instance**
  - Sometimes, it's useful to have a string representation of an instance of a class. To customize the string representation of a class instance, the class needs to implement the __str__ magic method
  - Internally, Python will call the __str__ method automatically when an instance calls the str() method
  - Note that the print() function converts all non-keyword arguments to strings by passing them to the str() before displaying the string values
  - The magic method __str__() is overridden to return a printable string representation of any user defined class
  - We have seen str() built-in function which returns a string from the object parameter. For example, str(12) returns '12'. When invoked, it calls the __str__() method in the int class

```
num = 12
print(type(str(num))) # <class 'str'>
```

```
# same as above
print((type(int.__str__(num)))) # <class 'str'>
```

- Let's see another **Example** with the Person class:

```
class Person:
  def __init__(self, first_name, last_name, age):
    self.first_name = first_name
    self.last_name = last_name
    self.age = age
```

- The Person class has three instance attributes including first_name, last_name, and age
  - The following creates a new instance of the Person class and display it:

```
person = Person('John', 'Doe', 25)
print(person)
```

    - Output:

```
<__main__.Person object at 0x0000023CA16D13A0>
```

➢ **Another Example to illustrate how to implement the __str__ method in the Person class:**

```
class Person:
  def __init__(self, first_name, last_name, age):
    self.first_name = first_name
    self.last_name = last_name
    self.age = age

  def __str__(self):
    return f'Person({self.first_name},{self.last_name},{self.age})'
```

- And when you use the print() function to print out an instance of the Person class, Python calls the __str__ method defined in the Person class. For example:

```
person = Person('John', 'Doe', 25)
print(person)
```

- Output:

```
Person(John,Doe,25)
```

✯ **Summary:**

- **Implement the __str__ method to customize the string representation of an instance of a class:**

```
class Lidl:
  """Printing instance of a class without __str__ """
  name = "John"

object_1 = Lidl()
print(object_1) # <__main__.Lidl object at 0x000001D59A739D20>

class Lidl:
  """Printing instance of a class with __str__ """
  name = "John"
  def __str__(self):
    return f"{Lidl.name}"

object_1 = Lidl()
print(object_1) # John

class Lidl:
  """Printing instance of a class without __str__ using class instance"""
  name = "John"

object_1 = Lidl()
print(object_1.name) # John

# So __str__ will print all attributes instead of writing manually for each instance
```

- override the __str__() method in the Employee class to return a string representation of its object:

```
class Employee:
  def __init__(self):
    self.name='Swati'
    self.salary=10000
  def __str__(self):
    return 'name='+self.name+' salary=$'+str(self.salary)
```

```
e1=Employee()
print(e1) # name=Swati salary=$10000
```

- See how the `str()` function internally calls the `__str__()` method defined in the Employee class. This is why it is called a magic method!

__repr__() method:

- **Below we are going to make sure that automatically we will have the values of what we instantiated included in a list then access any of those elements by using either for loop or __repr__**

```
class Item():
    pay_rate  = 0.8 # class attribute # The pay rate after 20% discount
    all = [] # we ill add or intances here

    def __init__(self, name: str, price: float, quantity=0):
        assert price >= 0, f"Price {price} is not greater or equal to zero"
        assert quantity >= 0, f"Price {quantity} is not greater or equal to zero"

        # assign to self object
        self.name = name
        self.price = price
        self.quantity = quantity

        # Actions to execute
        Item.all.append(self) # automatically the instances will fill our empty list

    def calculate_total_price(self):
        return self.price * self.quantity

    def apply_discount(self):
        self.price = self.price * self.pay_rate # self.pay_rate it has self in front because pay_rate can either be accessed from class level or instance level, therefore this will read from Item level (class level)

    def __repr__(self):
        return f"Item('{self.name}',{self.price},{self.quantity})" # so write it like you are writing when you instantiated the objects

# 5 instance created, so shop will be larger in the future to have more items
# the more items the more filtration you want to do in the future
item1 = Item("Phone" , 100, 1)
item2 = Item("Laptop" , 1000, 3)
item3 = Item("Cable" , 10, 5)
item4 = Item("Mouse" , 50, 5)
item5 = Item("Keyboard" , 75, 5)

print(Item.all) # we will have 5 elements

# we can access the stored items through for loops
# you can use filters to filter function to apply special things to the instances that they are matching your criteria
# we can use Lambda function inside the filters also once we try to use filters
for instance in Item.all:
    print("device is:",instance.name, "with the price of:",instance.price,"and quantity:",instance.quantity)

# OR you can use the magic method __repr__ (representing your objects):
print(Item.all)
```

f) Controlling Attribute Access:

- Many people coming to Python from other languages complain that it lacks true encapsulation for classes; that is, there's no way to define private attributes with public getter and setters
- This couldn't be farther than the truth: it just happens that Python accomplishes a great deal of encapsulation through "magic", instead of explicit modifiers for methods or fields. Take a look:

`__getattr__(self, name)`:
You can define behavior for when a user attempts to access an attribute that doesn't exist (either at all or yet),
this can be useful for catching and redirecting common misspellings, giving warnings about using deprecated attributes (you can still choose to compute and return that attribute, if you wish),
or deftly handing an `AttributeError`. It only gets called when a nonexistent attribute is accessed, however, so it isn't a true encapsulation solution

`__setattr__(self, name, value)`:
Unlike `__getattr__`, `__setattr__` is an encapsulation solution. It allows you to define behavior for assignment to an attribute regardless of whether or not that attribute exists,
meaning you can define custom rules for any changes in the values of attributes. However, you have to be careful with how you use `__setattr__`, as the example at the end
of the list will show

`__delattr__(self, name)`:
This is the exact same as `__setattr__`, but for deleting attributes instead of setting them. The same precautions need to be taken as with `__setattr__` as well in order to prevent infinite
recursion (calling `del self.name` in the implementation of `__delattr__` would cause infinite recursion).

`__getattribute__(self, name)`:
After all this, `__getattribute__` fits in pretty well with its companions `__setattr__` and `__delattr__`. However, I don't recommend you use it. `__getattribute__` can only be used with new-

style classes (all classes are new-style in the newest versions of Python, and in older versions you can make a class new-style by subclassing object. It allows you to define rules for whenever an attribute's value is accessed. It suffers from some similar infinite recursion problems as its partners-in-crime (this time you call the base class's __getattribute__ method to prevent this). It also mainly obviates the need for __getattr__, which, when __getattribute__ is implemented, only gets called if it is called explicitly or an AttributeError is raised. This method can be used (after all, it's your choice), but I don't recommend it because it has a small use case (it's far more rare that we need special behavior to retrieve a value than to assign to it) and because it can be really difficult to implement bug-free.
You can easily cause a problem in your definitions of any of the methods controlling attribute access. Consider this example:

```
def __setattr__(self, name, value):
    self.name = value
    # since every time an attribute is assigned, __setattr__() is called, this
    # is recursion.
    # so this really means self.__setattr__('name', value). Since the method
    # keeps calling itself, the recursion goes on forever causing a crash

def __setattr__(self, name, value):
    self.__dict__[name] = value # assigning to the dict of names in the class
    # define custom behavior here
```

- Again, Python's magic methods are incredibly powerful, and with great power comes great responsibility. It's important to know the proper way to use magic methods so you don't break any code
- So, what have we learned about custom attribute access in Python? It's not to be used lightly. In fact, it tends to be excessively powerful and counter-intuitive. But the reason why it exists is to scratch a certain itch: Python doesn't seek to make bad things impossible, but just to make them difficult. Freedom is paramount, so you can really do whatever you want. Here's an example of some of the special attribute access methods in action (note that we use super because not all classes have an attribute __dict__):

```
class AccessCounter(object):
    '''A class that contains a value and implements an access counter.
    The counter increments each time the value is changed.'''

    def __init__(self, val):
        super(AccessCounter, self).__setattr__('counter', 0)
        super(AccessCounter, self).__setattr__('value', val)

    def __setattr__(self, name, value):
        if name == 'value':
            super(AccessCounter, self).__setattr__('counter', self.counter + 1)
        # Make this unconditional.
        # If you want to prevent other attributes to be set, raise AttributeError(name)
        super(AccessCounter, self).__setattr__(name, value)

    def __delattr__(self, name):
        if name == 'value':
            super(AccessCounter, self).__setattr__('counter', self.counter + 1)
        super(AccessCounter, self).__delattr__(name)
```

g) Making Custom Sequences:

- There's a number of ways to get your Python classes to act like built in sequences (dict, tuple, list, str, etc.)
- These are by far my favorite magic methods in Python because of the absurd degree of control they give you and the way that they magically make a whole array of global functions work beautifully on instances of your class. But before we get down to the good stuff, a quick word on requirements
  - ***Requirements:***
    - Now that we're talking about creating your own sequences in Python, it's time to talk about *protocols*. Protocols are somewhat similar to interfaces in other languages in that they give you a set of methods you must define. However, in Python protocols are totally informal and require no explicit declarations to implement. Rather, they're more like guidelines
    - Why are we talking about protocols now? Because implementing custom container types in Python involves using some of these protocols. First, there's the protocol for defining immutable containers: to make an immutable container, you need only define __len__ and __getitem__ (more on these later). The mutable container protocol requires everything that immutable containers require plus __setitem__ and __delitem__. Lastly, if you want your object to be iterable, you'll have to define __iter__, which returns an iterator. That iterator must conform to an iterator protocol, which requires iterators to have methods called __iter__(returning itself) and next.
  - ***The magic behind containers:***
    - Without any more wait, here are the magic methods that containers use:

__len__(self):
Returns the length of the container. Part of the protocol for both immutable and mutable containers

__getitem__(self, key):
Defines behavior for when an item is accessed, using the notation self[key]. This is also part of both the mutable and immutable container protocols. It should also raise appropriate exceptions: TypeError if the type of the key is wrong and KeyError if there is no corresponding value for the key

__setitem__(self, key, value):
Defines behavior for when an item is assigned to, using the notation self[nkey] = value. This is part of the mutable container protocol. Again, you should

raise KeyError and TypeError where appropriate

__delitem__(self, key):
Defines behavior for when an item is deleted (e.g. del self[key]). This is only part of the mutable container protocol. You must raise the appropriate exceptions when an invalid key is used

__iter__(self):
Should return an iterator for the container. Iterators are returned in a number of contexts, most notably by the iter() built in function and when a container is looped over using the form for x in container:. Iterators are their own objects, and they also must define an __iter__ method that returns self

__reversed__(self):
Called to implement behavior for the reversed() built in function. Should return a reversed version of the sequence. Implement this only if the sequence class is ordered, like list or tuple

__contains__(self, item):
__contains__ defines behavior for membership tests using in and not in. Why isn't this part of a sequence protocol, you ask? Because when __contains__ isn't defined, Python just iterates over the sequence and returns True if it comes across the item it's looking for

__missing__(self, key):
__missing__ is used in subclasses of dict. It defines behavior for whenever a key is accessed that does not exist in a dictionary (so, for instance, if I had a dictionary d and said d["george"] when "george" is not a key in the dict, d.__missing__("george") would be called)

- ***An example:***
  - For our example, let's look at a list that implements some functional constructs that you might be used to from other languages (Haskell, for example).

```
class FunctionalList:
  '''A class wrapping a list with some extra functional magic, like head,
  tail, init, last, drop, and take.'''

  def __init__(self, values=None):
    if values is None:
      self.values = []
    else:
      self.values = values

  def __len__(self):
    return len(self.values)

  def __getitem__(self, key):
    # if key is of invalid type or value, the list values will raise the error
    return self.values[key]

  def __setitem__(self, key, value):
    self.values[key] = value

  def __delitem__(self, key):
    del self.values[key]

  def __iter__(self):
    return iter(self.values)

  def __reversed__(self):
    return reversed(self.values)

  def append(self, value):
    self.values.append(value)
  def head(self):
    # get the first element
    return self.values[0]
  def tail(self):
    # get all elements after the first
    return self.values[1:]
  def init(self):
    # get elements up to the last
    return self.values[:-1]
  def last(self):
    # get last element
    return self.values[-1]
  def drop(self, n):
    # get all elements except first n
    return self.values[n:]
  def take(self, n):
    # get first n elements
```

```
        return self.values[:n]
```

- There you have it, a (marginally) useful example of how to implement your own sequence. Of course, there are more useful applications of custom sequences, but quite a few of them are already implemented in the standard library (batteries included, right?), like Counter, OrderedDict, and NamedTuple.

h) Reflection:

- You can also control how reflection using the built in functions isinstance() and issubclass()behaves by defining magic methods. The magic methods are:

__instancecheck__(self, instance):
Checks if an instance is an instance of the class you defined (e.g. isinstance(instance, class)

__subclasscheck__(self, subclass):
Checks if a class subclasses the class you defined (e.g. issubclass(subclass, class))

- The use case for these magic methods might seem small, and that may very well be true. I won't spend too much more time on reflection magic methods because they aren't very important, but they reflect something important about object-oriented programming in Python and Python in general: there is almost always an easy way to do something, even if it's rarely necessary
- These magic methods might not seem useful, but if you ever need them you'll be glad that they're there

i) Callable Objects:

- As you may already know, in Python, functions are first-class objects. This means that they can be passed to functions and methods just as if they were objects of any other kind This is an incredibly powerful feature
- A special magic method in Python allows instances of your classes to behave as if they were functions, so that you can "call" them, pass them to functions that take functions as arguments, and so on
- This is another powerful convenience feature that makes programming in Python that much sweeter

__call__(self, [args...]):
Allows an instance of a class to be called as a function. Essentially, this means that x() is the same as x.__call__(). Note that __call__ takes a variable number of arguments; this means that you define __call__ as you would any other function, taking however many arguments you'd like it to.

__call__
can be particularly useful in classes with instances that need to often change state. "Calling" the instance can be an intuitive and elegant way to change the object's state. An example might be a class representing an entity's position on a plane:

```
class Entity:
    '''Class to represent an entity. Callable to update the entity's position.'''

    def __init__(self, size, x, y):
        self.x, self.y = x, y
        self.size = size

    def __call__(self, x, y):
        '''Change the position of the entity.'''
        self.x, self.y = x, y

    # snip...
```

j) Context Managers:

- In Python 2.5, a new keyword was introduced in Python along with a new method for code reuse: the with statement. The concept of context managers was hardly new in Python (it was implemented before as a part of the library), but not until PEP 343 was accepted did it achieve status as a first-class language construct. You may have seen with statements before:

```
with open('foo.txt') as bar:
    # perform some action with bar
```

- Context managers allow setup and cleanup actions to be taken for objects when their creation is wrapped with a with statement. The behavior of the context manager is determined by two magic methods:

__enter__(self):
Defines what the context manager should do at the beginning of the block created by the with statement. Note that the return value of __enter__ is bound to the *target* of the with statement, or the name after the as

__exit__(self, exception_type, exception_value, traceback):
Defines what the context manager should do after its block has been executed (or terminates). It can be used to handle exceptions, perform cleanup, or do something always done immediately after the action in the block. If the block executes successfully, exception_type, exception_value, and traceback will be None. Otherwise, you can choose to handle the exception or let the user handle it; if you want to handle it, make sure __exit__ returns True after all is said and done. If you don't want the exception to be handled by the context manager, just let it happen

__enter__ and __exit__
can be useful for specific classes that have well-defined and common behavior for setup and cleanup. You can also use these methods to create generic context managers that wrap other objects.Here's an example:

```
class Closer:
    '''A context manager to automatically close an object with a close method
    in a with statement.'''

    def __init__(self, obj):
        self.obj = obj
```

```
    def __enter__(self):
      return self.obj # bound to target

    def __exit__(self, exception_type, exception_val, trace):
      try:
        self.obj.close()
      except AttributeError: # obj isn't closable
        print 'Not closable.'
        return True # exception handled successfully
```

Here's an example of `Closer` in action, using an FTP connection to demonstrate it (a closable socket):

```
>>> from magicmethods import Closer
>>> from ftplib import FTP
>>> with Closer(FTP('ftp.somesite.com')) as conn:
...     conn.dir()
...
# output omitted for brevity
>>> conn.dir()
# long AttributeError message, can't use a connection that's closed
>>> with Closer(int(5)) as i:
...     i += 1
...
Not closable.
>>> i
6
```

▸ See how our wrapper gracefully handled both proper and improper uses? That's the power of context managers and magic methods. Note that the Python standard library includes a module contextlib that contains a context manager, `contextlib.closing()`, that does approximately the same thing (without any handling of the case where an object does not have a `close()` method)

k) Abstract Base Classes:

See http://docs.python.org/2/library/abc.html

l) Building Descriptor Objects:

- Descriptors are classes which, when accessed through either getting, setting, or deleting, can also alter other objects. Descriptors aren't meant to stand alone; rather, they're meant to be held by an owner class
- Descriptors can be useful when building object-oriented databases or classes that have attributes whose values are dependent on each other. Descriptors are particularly useful when representing attributes in several different units of measurement or representing computed attributes (like distance from the origin in a class to represent a point on a grid)
- To be a descriptor, a class must have at least one of `__get__`, `__set__`, and `__delete__` implemented. Let's take a look at those magic methods:

`__get__(self, instance, owner)`:
Define behavior for when the descriptor's value is retrieved. `instance` is the instance of the owner object. `owner` is the owner class itself

`__set__(self, instance, value)`:
Define behavior for when the descriptor's value is changed. `instance` is the instance of the owner class and `value` is the value to set the descriptor to

`__delete__(self, instance)`:
Define behavior for when the descriptor's value is deleted. `instance` is the instance of the owner object

Now, an example of a useful application of descriptors: unit conversions:

```
class Meter(object):
  '''Descriptor for a meter.'''

  def __init__(self, value=0.0):
    self.value = float(value)
  def __get__(self, instance, owner):
    return self.value
  def __set__(self, instance, value):
    self.value = float(value)

class Foot(object):
  '''Descriptor for a foot.'''

  def __get__(self, instance, owner):
    return instance.meter * 3.2808
  def __set__(self, instance, value):
    instance.meter = float(value) / 3.2808
```

```
class Distance(object):
    '''Class to represent distance holding two descriptors for feet and
    meters.'''
    meter = Meter()
    foot = Foot()
```

m) Copying:

- Sometimes, particularly when dealing with mutable objects, you want to be able to copy an object and make changes without affecting what you copied from. This is where Python's copy comes into play
- However (fortunately), Python modules are not sentient, so we don't have to worry about a Linux-based robot uprising, but we do have to tell Python how to efficiently copy things

`__copy__(self)`:
Defines behavior for `copy.copy()` for instances of your class. `copy.copy()` returns a *shallow copy* of your object -- this means that, while the instance itself is a new instance, all of its data is referenced -- i.e., the object itself is copied, but its data is still referenced (and hence changes to data in a shallow copy may cause changes in the original).

`__deepcopy__(self, memodict={})`:
Defines behavior for `copy.deepcopy()` for instances of your class. `copy.deepcopy()` returns a *deep copy* of your object -- the object *and* its data are both copied. `memodict` is a cache of previously copied objects -- this optimizes copying and prevents infinite recursion when copying recursive data structures. When you want to deep copy an individual attribute, call `copy.deepcopy()` on that attribute with `memodict` as the first argument

- What are some use cases for these magic methods? As always, in any case where you need more fine-grained control than what the default behavior gives you. For instance, if you are attempting to copy an object that stores a cache as a dictionary (which might be large), it might not make sense to copy the cache as well -- if the cache can be shared in memory between instances, then it should be

n) Pickling Your Objects:

- If you spend time with other Pythonistas, chances are you've at least heard of pickling. Pickling is a serialization process for Python data structures, and can be incredibly useful when you need to store an object and retrieve it later (usually for caching). It's also a major source of worries and confusion
- Pickling is so important that it doesn't just have its own module (`pickle`), but its own *protocol* and the magic methods to go with it. But first, a brief word on how to pickle existing types(feel free to skip it if you already know)
- Pickling: A Quick Soak in the Brine
  - Let's dive into pickling. Say you have a dictionary that you want to store and retrieve later. You couldwrite it's contents to a file, carefully making sure that you write correct syntax, then retrieve it using either `exec()` or processing the file input. But this is precarious at best: if you store important data in plain text, it could be corrupted or changed in any number of ways to make your program crash or worse run malicious code on your computer. Instead, we're going to pickle it:

```
import pickle

data = {'foo': [1, 2, 3],
        'bar': ('Hello', 'world!'),
        'baz': True}
jar = open('data.pkl', 'wb')
pickle.dump(data, jar) # write the pickled data to the file jar
jar.close()
```

Now, a few hours later, we want it back. All we have to do is unpickle it:

```
import pickle

pkl_file = open('data.pkl', 'rb') # connect to the pickled data
data = pickle.load(pkl_file) # load it into a variable
print data
pkl_file.close()
```

  - What happens? Exactly what you expect. It's just like we had data all along
  - Now, for a word of caution: pickling is not perfect. Pickle files are easily corrupted on accident and on purpose. Pickling may be more secure than using flat text files, but it still can be used to run malicious code. It's also incompatible across different versions of Python, so don't expect to distribute pickled objects and expect people to be able to open them. However, it can also be a powerful tool for caching and other common serialization tasks
    - Pickling your own Objects:
      - Pickling isn't just for built-in types. It's for any class that follows the pickle protocol. The pickle protocol has four optional methods for Python objects to customize how they act (it's a bit different for C extensions, but that's not in our scope):

`__getinitargs__(self)`:
If you'd like for `__init__` to be called when your class is unpickled, you can define `__getinitargs__`, which should return a tuple of the arguments that you'd like to be passed to `__init__`. Note that this method will only work for old-style classes.

`__getnewargs__(self)`:
For new-style classes, you can influence what arguments get passed to `__new__` upon unpickling. This method should also return a tuple of arguments that will then be passed to `__new__`.

`__getstate__(self)`:
Instead of the object's `__dict__` attribute being stored, you can return a custom state to be stored when the object is pickled. That state will be used by `__setstate__` when the object is unpickled.

`__setstate__(self, state)`:
When the object is unpickled, if `__setstate__` is defined the object's state will be passed to it instead of directly applied to the object's `__dict__`. This goes hand in hand with `__getstate__`: when both are defined, you can represent the object's pickled state however you want with whatever you want

__reduce__(self):
When defining extension types (i.e., types implemented using Python's C API), you have to tell Python how to pickle them if you want them to pickle them. __reduce__() is called when an object defining it is pickled. It can either return a string representing a global name that Python will look up and pickle, or a tuple. The tuple contains between 2 and 5 elements: a callable object that is called to recreate the object, a tuple of arguments for that callable object, state to be passed to __setstate__ (optional), an iterator yielding list items to be pickled (optional), and an iterator yielding dictionary items to be pickled (optional)

__reduce_ex__(self):

__reduce_ex__
exists for compatibility. If it is defined, __reduce_ex__ will be called over __reduce__ on pickling. __reduce__ can be defined as well for older versions of the pickling API that did not support __reduce_ex__.

An Example

Our example is a Slate, which remembers what its values have been and when those values were written to it. However, this particular slate goes blank each time it is pickled: the current value will not be saved.

```
import time

class Slate:
   '''Class to store a string and a changelog, and forget its value when
   pickled.'''

   def __init__(self, value):
      self.value = value
      self.last_change = time.asctime()
      self.history = {}

   def change(self, new_value):
      # Change the value. Commit last value to history
      self.history[self.last_change] = self.value
      self.value = new_value
      self.last_change = time.asctime()

   def print_changes(self):
      print 'Changelog for Slate object:'
      for k, v in self.history.items():
         print '%s\t %s' % (k, v)

   def __getstate__(self):
      # Deliberately do not return self.value or self.last_change.
      # We want to have a "blank slate" when we unpickle.
      return self.history

   def __setstate__(self, state):
      # Make self.history = state and last_change and value undefined
      self.history = state
      self.value, self.last_change = None, None
```

u) Changes in Python 3:

• Here, we document a few major places where Python 3 differs from 2.x in terms of its object model:

- Since the distinction between string and unicode has been done away with in Python 3, __unicode__ is gone and __bytes__ (which behaves similarly to __str__ and __unicode__ in 2.7) exists for a new built-in for constructing byte arrays.
- Since division defaults to true division in Python 3, __div__ is gone in Python 3
- __coerce__ is gone due to redundancy with other magic methods and confusing behavior
- __cmp__ is gone due to redundancy with other magic methods
- __nonzero__ has been renamed to __bool__

** THE BIG 7 (ALGORITHM) **

concept	parameters	algorithm	python
new variable	***name*: what to call it** ***type*: type of data** ***initVal*: starting value**	**create a variable called *name* of type *type* with initial value *initVal***	`name initVal`
output	***message*: text to write**	**output the text *message***	`print(message)`
input	***variable*: where answer will be stored** ***message*: question being asked**	**ask the user *message* and store result in**	`variable =` `input("message")`
convert to integer	***oldVariable*: in a non-integer format** ***intVariable*: integer to hold results**	**convert *oldVariable* to integer and store in *intVariable***	`intVariable =` `int(oldVariable)`
branch	***condition*: true / false value** ***code*: one or more lines of grouped code**	**if *condition* is true, execute *code***	`if condition:` `  code`

while loop	***sentry* - variable to control loop**
	***initialization code* - code to initialize sentry**
	***condition* - loop will repeat if condition is true**
	***change code* - code to change sentry so condition can become true**
		initialize sentry with *initialization code* then continue loop as long as *condition* is true. Inside loop, change *sentry* with *change code*	`initialization code`
			`while (condition):`
			`  code to repeat`
			`  change code`

for loop	***sentry* - integer variable to control loop** ***start* - integer starting value** ***finish* - integer ending value** ***change* - integer to add each pass**	**begin with *sentry* at *start* and add *change* to sentry on each pass until *sentry* is larger than or equal to *finish***	for *sentry* in range(*start*, *finish*, *change*): *code to repeat*

function	***functionName* - the name of the function**
	***return type* - type of data function returns. Could be nothing**
	***parameters* - one or more values to send to the function**
		create a function called *functionName* passing the values *parameters* that returns a value of *return type*	`def functionName():`
			`  code`
			`  return something`

1. CONCISE CODING:

- each line should solve only one thing when possible:
  x=parseInt(rand(a,process(b)))
  does a lot at once!
- this is the line that's going to break
- consider breaking it into several lines now because you'll have to when it does malfunction
- if you find yourself counting parentheses, it's probably too complicated (you can create temporary variables in a function that they will go after the function is used)
- fewer lines = more efficient code

2. OPTIMIZATION? NOT SO FAST!

- optimize for the programmer first
  -> the most inefficient part of modern coding is debugging
  -> make your code easy to understand first
  -> do not worry about maximizing time or space efficiency until your code is actually working
  -> **don't tune up an engine that isn't running** (because it's not running first of all, make it run first)
  -> **"premature optimization is the root of all evil"** - Donald Knuth, father of algorithm analysis

3. INDENTATION:

- Indentation is **NOT** optional, regardless of the language
- indent the content of every branch, loop, function, and class
- make sure you're 'outdents' match indents
- consider labelling your **}** characters with comments
- if you think you're done coding and you aren't back at the left margin, something is wrong
- python has issues with space or tab -> you need to be careful
- if you indentation to create block
- at company you will always have style guide to follow (ex. for indentation)

4. MORE INDENTATION:

- use your editor tools to ensure the blocks are what you think they are
- you can use tabs or spaces, but do not change in the same file
- I'm unconcerned about how wide your indentation is, as long as you are consistent. 2 or 4 spaces is common
- if indentation goes deeper than four or five levels, consider refactoring your code

5. VARIABLE NAMES:

- should generally be full words or phrases
- x and y should generally be used only as coordinates
- I can be a counting variable when there is no other better name
- variable names should indicate what the variable does or contains: frank, and theThing are not useful
- avoid abbreviations: userName is better than un or uName
- use specific names to avoid using keywords: userName rather than name (which is often reserved)

6. CAPITALIZATION:

- ordinary variables (including class instances) should begin with a lowercase letter
- Class names should begin with an uppercase letter
- use camel casing to combine long names: userName rather than username or user_name
- Constants or variables acting like constants should be all upperCase (NUM_VALUES)
- Constants can use underscores to separate words

7. SELF-DOCUMENTING VARIABLES:

- **type identifiers** are not required, but can be helpful especially in **GUI applications**: txtUsername is a text box containing the user name, and userName is the actual user name string.

- the best variable names lead to **self-documenting code**:
  if(guess==correct)
- array names can be plural, but individual values should not be

8. VARIABLES SCOPE:

- avoid global variables. Use parameters and return values to manage passing data if possible
- global constants are fine
- in **OOP** setting, class-level properties are fine, but should still be protected

9. FILE NAMES:

- file names should be single words. Spaces cause major problems for command-line compilers
- use standard conventions for file names in your project: .c .py, .h or whatever makes sense for your language
- files created by your program should also have standard extensions:
  - **-> .txt** for human-readable text
  - **-> .csv** for comma-separated values
  - **-> .dat** for binary files that don't have a more specific meaning
- you may be given specific file names to use. Failure to use these names exactly will result in automated systems not finding your code or data, and considering your work a failure

10. BRANCHING:

- avoid single-line if statements: include **{}** in even the simplest statements in languages that require it
- avoid **ternary operators** if possible. Expand into complete if-else structures when possible
- avoid **Boolean operators** in if statements (until you can explain **DeMorgan's law**):
  - -> use **if / elseif** for **OR**
  - -> use **nested if** for **and**
- so, if you find that you have a complicated if statement or loop that's not working correctly the vast majority of time if you break it up into smaller non-compound conditions, you're gonna get what you want

11. MORE BRANCHING:

- **switch / select statements** can cause a lot of grief. Consider using an **if/elseif** instead, or a function, dictionary, or array (**python does not switch**)
- always include a **final else** clause for a **multiple elseif** or **switch statement**, even if it is impossible for that else to occur. It still will.
- know the different types of **aggregate variable types** in your language (**lists, tuples, arrays, dictionaries**, etc) and **use the right tool for the job at hand**
- be careful with **equality operators**. Different languages have very different behaviors

12. PICKING A LOOP TYPE:

- use for **loops** for **deterministic repetition**: you know how many times something will happen before loop begins (it is about counting of some sort)
- for loops create less problems than while loops
- for **loops** (or **foreach or iterators**) are ideal for **iterating through a listable data structure**: use them when you can
- **while** loops are **non-deterministic**: based on **events or user input**

13. FOR LOOP SENTRIES:

- use **i** as a for **loop sentry** only if the name doesn't matter (**sentry variable is the variable that its job is to control access to that loop**)
- for double-nested loops you can use j for the inner loop
- if possible, pick a meaningful name (cardNumber) instead of i
- most for loops have ways to count backwards and skip count
- it may be possible to make a for loop act like a while loop
- resist the temptation (do not be crazy to make a for loop act like a while loop)

14. WHILE CONDITIONS:

- remember that the condition of a while loop indicates when you will **STAY** in the loop
- often what you really want to know is when you will **LEAVE** the loop
- you often must think of the opposite of what you're trying to say
- therefore, while loop conditions can be so tricky
- it's **MUCH** messier when you have compound conditions (**DeMorgan** again)

15. COMPOUND CONDITION TRAPS:

- if (myVar == "A" or "B") will **NOT** do what you think
- it will evaluate whether myVar is equal to "A"
- it will **NOT** evaluate whether myVar is equal to "B"
- in most languages, it will simply return true if "B" is a thing
- **compound conditions** are problematic again

16. THE FAMOUS 'keepGoing' LOOP:

- use a **Boolean sentry** - **true (1)** or **false (0)**
- **initialize it outside the loop (tell python that this var exist so we can go back and change it later)**
- change inside the loop in basic if statements
- this allows you to keep all the conditions positive
- it avoids issues with **DeMorgan's**
- **most of my loops:**
  while(keepGoing){
- **you can replace compound conditions with keepGoing loops**

17. BREAKING BAD:

- it is a sign that you are a lazy programmer
- the break statement is troublesome
- it is mostly used in poorly designed structures
  -> C++ switch
- it's something used in 'endless' loops
- ... that aren't really endless
- you know what I'm going to say...
  -> Boolean conditions

18. SINGLE-PURPOSE FUNCTIONS:

- a function should solve a very specific problem
  -> if it does more than one thing, consider making it more than one function
  -> any repeated or complex behavior that can be named is a good candidate for a function
  -> a function should ideally be small enough to see all at once in your editor
  -> it should have a reasonable (<10) number of local variables
  -> it's the space you can understand readily

19. THE POINT OF NO (OR ONE) RETURN:

- if a function does not return a value, it doesn't need an empty return statement
- return should typically be the last line of a function
- in most cases, a function can be re-written to only need one return statement
- make a variable, change its value, and return that variable
- recursion might be an exception

20. ADVANCED FUNCTION TECHNIQUES CAN WAIT:

- **inner functions, lambdas, and recursion are fine techniques**
  -> but use them only when they improve your code
  -> they can cause a lot of a confusion if not used well
- don't get fancy unless there's a good reason
  -> there usually isn't a good reason
- if a function gets too complex, consider a data-driven solution
- try to test a function in isolation before integrating it

21. CLASSES:

- an object should have a meaningful name
- every object should have a constructor or equivalent
  -> constructor should initialize all properties / attributes
- properties should be private (or protected) if possible
  -> use access modifiers (**getters and setters**) to allow access to properties
  -> consider using a **property mechanism** (java beans, python property decorator) **to make virtual properties**
- **use multiple inheritance only when required: consider using abstraction or aggregation instead**
- **consider adding unit tests to your classes. They can be formal, or simply a function which tests all the features of the class**

22. DEBUGGING:

- **Detective Mode = try to find a pattern**
- when the **program** is **working without error** but giving the **wrong outcome = bug -> what patterns I can find to see why it is not working**
- errors are normal part of programming, begin debugging, did you tell it what to do incorrectly? or did you tell it to do the wrong thing?
- **use of print statements for debugging is ok, but not ideal**
- diagnostic print statements should not be part of final code

- **use of a debugger is preferred when possible**
- **learn command-line debuggers just in case**
- in some cases, you will also need to use other tools like valgrind
- **most beginners assume it's an implementation problem**
- **usually, it's really an algorithm problem**
- **bad implementation can be googled**
- **what tools can you use?**
- **DON'T start with a solution**
- **start by truly understanding the problem**
- **debugging and data - replay in slow motion like in a football match**
- **pdb: python debugger**
- **put this line where you want to use debugger: pdb.set_trace() = STOP and go in debugger mode**
- **at PdB prompt enter help for commands**
- **list: will show the code and where the next line is going to run**
- **next: moves to the next line**
- **print: prints what value you want -> ex. print(i) or p for shorthand**
- **shorthand: l,p, n,q**

23. USING A BUILD TOOL:

- you might be required to use make or another build tool
  -> do **NOT** wait until the end to build your make file. It should be continuously updated
  -> your make file will generally need a run target. You may also be required to have other targets

24. USING VERSION CONTROL:

- so, **Git** allows you to make **save points** when your program runs, so make a **commit**
- **Git** or other **version control** is well worth learning
- make frequent commits (each time you get up or solve a smaller problem)
- consider making a **branch** whenever you have a runnable approximation of the code
- do **NOT** wait until the project is done to commit
- avoid using nested commits. Each project should be a single repo
- you may need to save the repo to a specific name so an automated system can find it

25. USING VERSION CONTROL:

-> just start with a simple game like tetris
-> start with C++, that's they use in industry
-> if you use visual basic, you don't have to code until the end
-> the best way to start is to pick a problem you want to solve
-> you'll stay motivated if you work on a real-world problem

26. WHAT I WISH I'D BEEN TAUGHT:

- **programming isn't about languages (the stupid question: what language should I start with?)**
- **python is not good in teaching direct memory management because it does it for us**
- **the language ultimately doesn't matter much**
- **there's not a lot of memorizing**
- **most programming isn't about math**
- **programming languages are simpler than human ones**

27. CODE ISN'T ABOUT LANGUAGE:

- **coding has only about eight concepts**
- **they work in almost the same way in every language**
- **the secret isn't code, the secret is algorithm and data**
- **learn how to use these concepts in English**
- **write out the concepts first, then convert to code later**
- **most beginners think they don't understand what code to write**
- **the real problem is they don't really understand the problem they're trying to solve**
- **experts do that too...**

28. COMMENTS ARE CODE:

- comments explain code to other programmers
- **NO!**
- **code explains the comments to the computer**

#1 Python tutorial for beginners
#2 variables
#3 multiple assignment
#4 string methods
#5 type cast
#6 user input
#7 math functions
#8 string slicing
#9 if statements
#10 logical operators
#11 while loops
#12 for loops
#13 nested loops
#14 break continue pass
#15 lists
#16 2D lists
#17 tuples
#18 sets
#19 dictionaries
#20 indexing
#21 functions
#22 return statement
#23 keyword arguments
#24 nested function calls
#25 variable scope

#26 *args
#27 **kwargs
#28 string format
#29 random numbers
#30 exception handling
#31 file detection
#32 read a file
#33 write a file
#34 copy a file
#35 move a file
#36 delete a file
#37 modules
#38 rock, paper, scissors game
#39 quiz game
#40 Object Oriented Programming (OOP)
#41 class variables
#42 inheritance
#43 multilevel inheritance
#44 multiple inheritance
#45 method overriding
#46 method chaining
#47 super function
#48 abstract classes
#49 objects as arguments
#50 duck typing

#51 walrus operator
#52 functions to variables
#53 higher order functions
#54 lambda λ
#55 sort
#56 map
#57 filter
#58 reduce
#59 list comprehensions
#60 dictionary comprehensions
#61 zip function
#62 if *name* == '__main__'
#63 time module
#64 threading
#65 daemon threads
#66 multiprocessing
#67 GUI windows
#68 labels
#69 buttons
#70 entrybox
#71 checkbox
#72 radio buttons
#73 scale
#74 listbox
#75 messagebox

#76 colorchooser
#77 text area
#78 open a file (file dialog)
#79 save a file (file dialog)
#80 menubar
#81 frames
#82 new windows
#83 window tabs
#84 grid
#85 progress bar
#86 canvas
#87 keyboard events
#88 mouse events
#89 drag & drop
#90 move images w/ keys
#91 animations
#92 multiple animations
#93 clock program
#94 send an email
#95 run with command prompt
#96 pip
#97 py to exe
#98 calculator program
#99 text editor program
100 tic tac toe game

101 snake game

102 overloading operators
103 method overloading - you can do with this or *args
104 optional parameters tutorial #1
105 private and public classes
106 static methods & class methods
107 static methods and class
108 unpacking
109 packages

1. Starting variables

2. If's (8):
   - 2.1. if/elif/else with (“in”, “and”, “or”, “not in”): "l" in "lulu"
   - 2.2. if/elif/else with Boolean (var, input, def): employed
   - 2.3. if/elif/else with condition: earn 200$
   - 2.4. use nested if's: variations where to go based on money in the pocket
   - 2.5. if with len(): username
   - 2.6. if with replace: dash in a word
   - 2.7. if with indexing: dash in word using index
   - 2.8. if with "is" and or "is not": compare 2 strings OR 2 tuples

3. While's (8):
   - 3.1. while with if, “in” and “or”: letter word," ","-"
   - 3.2. while with expression: guess word & lives
   - 3.3. while with comparison operator: curry spoons
   - 3.4. while with len(): username input instead of blank space
   - 3.5. use multiple while: days of the week
   - 3.6. while with "not" and "not in" using a Boolean variable and a list: check if input already chosen or not by user and that it does not overlap
   - 3.7. while with "else"
   - 3.8. while with "True" at the beginning and use break to end the loop at the end

4. For's (8):
   - 4.1. for one under each other from a list: menu below each other
   - 4.2. for one next to each other in a list:
     - 4.2.1. expression for item in iterable:
       - a. for str menu next to each other
       - b. for numbers: count 2 trees at a time
     - 4.2.2. expression if/else for item in iterable: compare 2 lists and change letter to "_"
     - 4.2.3. expression for item in iterable if conditional:
       - a. print a list of passing marks
       - b. print a list of passing marks and the rest failed
   - 4.3. for with direct tuple or list next to it: personal information
   - 4.4. for loop with index OR item:
     - a. index: print a range of students in a class of 20
     - b. item: check specific fruit in shopping cart and print that fruit if found
   - 4.5. for with var at index: highest mark from marks
   - 4.6. iterate with for loop through dictionary: capitals of countries
   - 4.7. use nested for: print symbol over rows and columns
   - 4.8. for with var set to 0
   - 4.9. iterate with for into a 2D list

5. Def's (12):
   - 5.1. FEED arguments from the caller to def parameters, so that the def WILL DO tasks for you:
     - a. generate a random number from 1 to a number given by user, and repeat that request for the user
     - b. add 2 numbers given by the user, and repeat that request for the user
   - 5.2. create functions to MAKE something:
     - 5.2.1. save value instead of printing it: ant marching using global and local variables by asking user for the ant number with a multi-line string return output

5.2.2. return 2 values in a function: add and subtract same numbers using same function

5.3. use keyword to pass arguments from caller to the parameter you want: pass arguments in the caller in different order than the parameters

5.4. change index of a list using def : create function to change index of a list

5.5. use * to pass how many arguments you want: create function to add any number

5.6. use default if you don't want to pass all arguments from the caller: make default parameter age and pass other arguments from caller

5.7. call function from another function: create def total and call 2 functions sum and subtract from here

5.8. pass from list from another module: bring a list of words, pass words as argument and create a random word once called random_word

5.9. put caller in a var and use it later: create a string in def and store it in a var, return, then the caller put it in a var and then show output

5.10. pass variable or list as argument through a function

5.11. mix of function with global variable, Boolean variable of other function and return: create board for tic tac toe, check rows if winner

5.12. use function arguments in other function parameters

6. Class's Integers (5):

6.1. creating a simple class

6.2. create a constructor

6.3. access methods in a class using getters and setters

6.4. create an inheritance

6.5. overload a method

7. Operations on Strings, Integers & various commands (11):

7.1. find out type

7.2. cast a spell

7.3. create f string, string interpolation and string formatting

7.4. print empty line, multi-line strings and next to each other

7.5. join a string

7.6. slice a salami

7.8. delay a command

7.9. create empty set, list, string, dictionary, tuple

7.10. convert 1 string to list -> to set -> to tuple -> to dictionary and do operations

7.11. convert list with more than 1 string to list -> to set -> to tuple -> to dictionary and do operations

Smart Training

1. we check in a patient named john Smith
he's 20 years old and is he's a new patient
create 3 variables for name, age and one that would advise if he's a new or old patient

2. ask two questions: person's name and favorite color, then, print a message like "Mosh like blue"

3. ask a user their weight (in pounds), convert it to kilograms and print on the terminal

4. what is the output for:
```
name = 'Jennifer'
print(name[1:-1])
```

5. price of a house is 1M
if buyer has a good credit, they need to put down 10%
otherwise, they need to put down 20%. Print the down payment

6. build the following using logical operators AND:
if applicant has high income AND good credit, Eligible for loan

7. build the following using logical operators NOT:
if applicant has good credit AND doesn't have a criminal record

8. use comparison operators for below:
if temperature is greater than 30, it's a hot day, otherwise, if it's less than 10 it's a cold day, otherwise it's neither hot or cold

9. if name is less than 3 characters long name must be at least 3 characters, otherwise, if it's more than 50 characters long name cannot be a maximum of 50 characters, otherwise name looks good, write a program to implement these rules

10. create a program that will ask:
weight:

(L)bs or (K)g:
print the result

11. create a game for guessing the number 9 in which you have only 3 attempts

12. create a car game that will do the following:
> **<---- will show this symbol**
> help **<-- if you type help then you will have below list of commands**
start - to start the car
stop - to stop the car
quit - to exit
> asd
I don't understand that
> start
Car started...Ready to go!
> stop
Car stopped.
> quit
program terminates, then add if car is running then advise that it started already same for stop

13. calculate the total of below shopping cart:
prices = [10,20,30]

14. print coordinates of x in range of 4 and y in range of 3 in format string

15. draw an F shape:
xxxxx
xx
xxxxx
xx
xx

16. print from the 2nd index of the list:
names = ['john','bob','mosh','sarah','mary']

17. write a program to find the largest number in a list

18. write a matrix of 3 by 3, print the second item in the list then iterate over all the items in the matrix

19. for the list: numbers = [5,2,1,7,5,4]
check if 50 is in the list, check how many times you have 5 in the list, check at what index the number 5 is, sort your list ascending then descending, copy the list to another var, back to the first list add number 20
insert number 10 at index 0, remove the number 5, remove the last item then clear everything and print numbers and the new variable numbers2

20. write a program to remove the duplicates in the following list: numbers = [2,2,4,6,3,6,1]

21. create a program that asks the phone number then prints in words
phone:1234
One Two Three Four

22. what is the result for x = (2 + 3) * 10 - 3

23. pack the following: coordinates = (1, 2, 3), then unpack first item

24. define the var "customer" with the below data, print the value associated with the "name" key, change the value of "name" to "jack Smith" and add the key "birthday" with the value 1980 and print it:
Name: John Smith
Email: john@gmail.com
Phone: 1234

25. create a program to print your message on the screen transforming **:) OR :(INTO** emjoji

26. calculate square of the number 3

27. reorganize the code from exercise 25 and extract a reusable function in this code, because this algorithm for converting smiley faces into emojis can be used in various applications,
advise from which line to which line belongs to the algorithm and explain why the rest does not belong to it

28. write a program to get the user age from the terminal then pass a string, take into account that user income as 20000
calculate the financial risk, handle errors if age is 0 and if user enters wrong input

29. define a new type called person, that should have a name:
- name attribute
- talk() methode
print talk() with the name of the first object then create another object named bob smith and print its attribute talk

30. take the exercise 17 and write a function called find_max, this function should take a list and return the largest number in that list,
extract and put that function in a different module called utils, import the utils in the current module and call this function then get result and print it on the terminal

```
numbers = [3, 6, 2, 8, 4, 10]
max = numbers[0]
for number in numbers:
  if number > max:
    max = number
print(max)
```

31. print 3 times a random floating point number in the range(0.01) and 1.0

32. write a program to roll a dice, define a class called Dice that will have a method called roll(), so every time
we call this method we get a tuple, so every time we call tuple, we should get 2 random values

33. which is the module in python to access files and directory?

34. reference a directory in your project, create a path object and you can pass a name of directory("ecommerce")

35. create a new directory names "emails" in current folder

36. remove created directory "emails" created previously

37. find all files and directory in current path for python or excel path and show on console

38. read from a text document inside python document

39. write to a text document inside a python folder

40. random number quiz game with message if too low or too high

41. quiz program without timer

41. quiz program with timer

42. Tic-Tac-Toe - Aaron Bernath

43. Tic-Tac-Toe players using inheritance implementation - Kylie Ying

44. clock program – Bro Code

45. rock paper scissors game – Bro Code

46. rock paper scissors game – Kylie Ying

47. madlibs – Kylie Ying

48. guess the number – Kylie Ying

49. hangman – Kylie Ying

50. create a car object, with attributes: model, year, and color and with the methods drive (“the car is driving”) and stop (“this car is stopped”)
create 2 instances of that object, car_1 and car_2

51. add a class variable of 4 wheels for exercise 51

52. create an animal object with methods “eat” and “sleep” that will be inherited by other animals

53. create an object “organism” that will be inherited by an animal, and that animal will be inherited by a dog

54. create 3 objects rabbit, hawk, and fish, all 3 of them will be pray but only the hawk and fish will be predator

55. create an object of an animal that will have attribute to eat but then another object linked to the animal object as rabbit that will have same attribute but to eat carrot

56. create a car object that once called will start, break, and stop automatically

57. create an object user of class User with the arguments ("Dave Bowman","19710315"), first extract the first and last name and calculate the age by extracting the age from the birthday

1.
```
full_name = "John Smith"
age = 20
is_new = False
```

2.
```
name = input('What is your name ')
favorite_color = input("what is your favorite color? ")
print(name + ' likes ' + favorite_color)
```

3.
```
weight_in_lbs = input("please insert your weight in (lbs) ")
weight_in_kg = int(weight_in_pounds) * 0.45
print(weight_in_kg)
```

4.
```
ennife
```

5.
```
price = 1000000
has_good_credit = True
if has_good_credit:
  down_payment =  0.10 * price
else:
  down_payment = 0.20 * price
print(f"down payment will be ${down_payment}")
```

6.
```
has_high_income = True
has_good_credit = True
if has_high_income and has_good_credit:
  print("Eligible for loan")
else:
  print("Not eligible for a loan")
```

7.
```
has_good_credit = True
```

```
has_criminal_record = False
if has_good_credit and not has_criminal_record:
  print("Eligible for loan")
else:
  print("Not eligible for a loan")
```

8.

```
temperature = int(input("please enter the temperature: "))
if temperature > 30:
  print("it's a hot day")
elif temperature < 10:
  print("it's a cold day")
else:
  print("it's neither hot or cold")
```

9.

```
name = input("please insert username: ")
if len(name) < 3:
  print("Name must be at least 3 characters")
elif len(name) > 50:
  print("Name can be a maximum of 50 characters")
else:
  print("Name looks good!")
```

10.

```
weight = int(input('Weight: '))
unit = input('(L)bs or (K)g: ')
if unit.upper() == "L":
  converted = weight * 0.45
  print(f"You are {converted} kilos")
else:
  converted = weight / 0.45
  print(f"You are {converted} pounds")
```

11.

```
secret_number = 9
guess_count = 0
guess_limit = 3
while guess_count < guess_limit:
  guess = int(input('Guess: '))
  guess_count += 1
  if guess == secret_number:
    print("You won")
    break
else:
  print("You have lost")
```

12.

```
command = ""
started = False
while True:
  command = input("> ").lower()
  if command == "start":
    if started:
      print("Car already started")
    else:
      started = True
      print("Car started...")
  elif command == "stop":
    if not started:
      print("Car is already stopped")
    else:
```

```
        started = False
        print("Car stopped.")
    elif command == "help":
      print("""
start - to start the car
stop - to stop the car
quit - to quit
      """)
    elif command == "quit":
      break
    else:
      print("sorry, I don't understand that")
```

13.

Use for loop

```
prices = [10,20,30]
total = 0
for price in prices:
  total += price
print(f"Total: {total}")
```

14.

```
for x in range(4):
  for y in range(3):
    print(f'({x}, {y})')
```

15.

```
number = [5, 2, 5, 2, 2]
for x_count in number:
  output = ' '
  for count in range(x_count):
    output += 'x'
  print(output)
```

16.

```
names = ['john','bob','mosh','sarah','mary']
print(names[2:])
```

17.

```
numbers = [3, 6, 2, 8, 4, 10]
max = numbers[0]
for number in numbers:
  if number > max:
    max = number
print(max)
```

18.

```
matrix = [ <- use 2 dimensional list
[1,2,3],
[4,5,6],
[7,8,9]
]
print(matrix)
print(matrix[0][1])
for row in matrix: # <- use nested loops
  for item in row:
    print(item)
```

19.

```
numbers = [5,2,1,7,4]
print(50 in numbers)
numbers.count(5)
```

```
numbers.index(5)
numbers.sort()
numbers.reverse()
numbers2 = numbers.copy()
numbers.append(20)
numbers.insert(0,10)
numbers.remove(5)
numbers.pop()
numbers.clear()
print(numbers)
print(numbers2)
```

20.

```
numbers = [2,2,4,6,3,6,1]
uniques = []
for number in numbers:
  if number not in uniques:
    uniques.append(number)
print(uniques)
```

21.

```
phone = input("Phone :")
digits_mapping = {
"1":"One",
"2":"Two",
"3":"Three",
"4":"Four",
"5":"Five",
"6":"Six",
"7":"Seven",
"8":"Eight",
"9":"Nine",
"10":"Ten"
}
output = ""
for ch in phone:
  output += digits_mapping.get(ch,"!") + " "
print(output)
```

22.

output: 47 **(operator precedence)**

23.

```
coordinates = (1, 2, 3) # works for list also [1, 2, 3]
x,y,z = coordinates # packing
print(x) # unpacking
```

24.

use dictionary

```
customer = {
   "name":"john smith",
   "age":30,
   "is_verified":True
}
print(customer.get("Name"))
customer["name"] = "jack smith"
customer["birthday"] = "1980"
print(customer["birthday"])
```

25.

use dictionary and map key value pair and map it to a string that you add in an emoji

```
message = input(">")
```

```
words = message.split(' ')
emojis = {
    ":)":":)",
    ":(":":("
}
output = ""
for word in words:
 output += emojis.get(word, word) + " "
print(output)
```

26.

```
square of a number = multiply the given number by itself
def square(number):
return number * number # <- to return this number outside of this function we use the return statement
                       # <- return statement will return values to the callers of our function, it's like input() that takes the user input and returns it as a string => it will return a value
                       # -> will return this number outside of this function and we can store it if we want in a variable like we do with input() => return statement behave same as input()
result = square(3)
print(result)
```

27.

```
# the line words till the line output belongs to algorithm as input can come in different forms, we can receive it from the terminal, graphical user interface (GUI) and this line will not be reusable
# we need to pass the message as a string, but we don't care how we got that message
# we shouldn’t include print(output) as what we do with the output is different from one program to another, in this program we will print it in other we will send the output as an email or as a response in a chat application
# so, my function in general should not worry about receiving an output or printing it, these lines of code should not belong to a function
# the name of the function should clearly explain what task it does, and the function has to do only one task
def emoji_converter(message):
 words = message.split(' ')
 emojis = {
    ":)":":)",
    ":(":":("
 }
 output = ""
 for word in words:
  output += emojis.get(word, word) + " "
 return output
message = input(">")
result = emoji_converter(message)
print(result)
```

28.

```
use exceptions (use try-except blocks, anticipate them and handle them)
- try means "I’m going to do something dangerous", so try command acknowledges that next block of code could be potentially dangerous
- you will use a safety net = except clause = "if you are about to crash, don't crash!, come to me and I will give you what to do"
- we can have multiple exception handling if more stuff could go wrong
- will be used when you will have database and files
try:
 age = int(input('Age: '))
 income = 20000
 risk = income / age
 print(age)
except ZeroDivisionError:
 print('age cannot be 0')
except ValueError:
 print('Invalid value')
```

29.

```
class Person:
 def __init__(self, name):
  self.name = name
 def talk(self):
  print(f"Hi my name is {self.name}")
lulu = Person("lulu")
```

```
lulu.talk()
bob = Person("Bob Smith")
bob.talk()
```

30.

```
utils.py
def find_max(numbers):
    max = numbers[0]
    for number in numbers:
        if number > max:
            max = number
    return max

app.py
from utils import find_max
numbers = [10,3,6,2]
maximum = find_max(numbers)
print(maximum)
```

31.

```
# use python built in modules, in this case random
import random
for i in range(3):
  print(random.randint(10,20))
```

32.

```
import random
class Dice:
  def roll(self):
    first = random.randint(1,6)
    secound = random.randint(1,6)
    return first,secound
dice = Dice()
print(dice.roll())
```

33.

```
pathlib
```

34.

```
from pathlib import Path
path = Path("ecommerce")
print(path.exists())
```

35.

```
from pathlib import Path
path = Path("emails")
print(path.mkdir()) <- does not return value but creates the directory
```

36.

```
from pathlib import Path
path = Path("emails")
print(path.rmdir())
```

37.

```
from pathlib import Path
path = Path()
for file in (path.glob('*.py'))  <- * means everything, add an extension with *.* or *.py or *.xls
  print(file)
```

38.

```
from pathlib import Path
with open('c:\\python\\notes.txt') as file:
```

```
print(file.read())
```

39.

```
from pathlib import Path
text = "this text was added to the file in python"
with open('c:\\python\\notes.txt','a')as file:
    file.write(text)
```

40.

```
import random

def random_number_game(x):
    random_number = random.randint(1,x)
    guess = 0

    while guess != random_number:
        guess = int(input(f"please enter a number from 1 to {x}: "))
        if guess > random_number:
            print("too high")
        elif guess < random_number:
            print("too low")
    print(f"congrats you have won, the magic number was {random_number}")

random_number_game(10)
```

41.

```
# -------------------------
def new_game():

    guesses = []
    correct_guesses = 0
    question_num = 1

    for key in questions:
        print("-------------------------")
        print(key)
        for i in options[question_num-1]:
            print(i)
        guess = input("Enter (A, B, C, or D): ")
        guess = guess.upper()
        guesses.append(guess)

        correct_guesses += check_answer(questions.get(key), guess)
        question_num += 1

    display_score(correct_guesses, guesses)

# -------------------------
def check_answer(answer, guess):

    if answer == guess:
        print("CORRECT!")
        return 1
    else:
        print("WRONG!")
        return 0

# -------------------------
def display_score(correct_guesses, guesses):
    print("-------------------------")
    print("RESULTS")
    print("-------------------------")

    print("Answers: ", end="")
```

```
    for i in questions:
        print(questions.get(i), end=" ")
    print()

    print("Guesses: ", end="")
    for i in guesses:
        print(i, end=" ")
    print()

    score = int((correct_guesses/len(questions))*100)
    print("Your score is: "+str(score)+"%")

# -----------------------
def play_again():

    response = input("Do you want to play again? (yes or no): ")
    response = response.upper()

    if response == "YES":
        return True
    else:
        return False
# -----------------------


questions = {
 "Who created Python?: ": "A",
 "What year was Python created?: ": "B",
 "Python is tributed to which comedy group?: ": "C",
 "Is the Earth round?: ": "A"
}

options = [["A. Guido van Rossum", "B. Elon Musk", "C. Bill Gates", "D. Mark Zuckerburg"],
          ["A. 1989", "B. 1991", "C. 2000", "D. 2016"],
          ["A. Lonely Island", "B. Smosh", "C. Monty Python", "D. SNL"],
          ["A. True","B. False", "C. sometimes", "D. What's Earth?"]]

new_game()

while play_again():
    new_game()

print("Byeeeeee!")

# -----------------------
```

42.

```
import time
import threading
score = 0
def countdown():
  global my_timer
  my_timer = 10
  for x in range(10):
    my_timer = my_timer - 1
    time.sleep(1.0)
  print("out of time")
countdown_thread = threading.Thread(target = countdown)
countdown_thread.start()
while my_timer > 0:
  name = input("What is your name: ")
  print("GENERAL KNOWLEDGE")
```

```
 print("Question 1:")
 time.sleep(1.0)
 print("Which planet is furthest from the sun?")
 time.sleep(1.0)
 print("a. Jupiter")
 time.sleep(0.5)
 print("b. Earth")
 time.sleep(0.5)
 print("c. Neptune")
 time.sleep(0.5)
 Answer = input("Please enter your password:" )
 if Ansawer =="c":
  print("Correct")
  score = score + 10
  print(name,"your score is",score)
 else:
  print("Incorrect")
  print("Your score is", score)
 print()
 time.sleep(2.0)
 if my_timmer == 0:
  break
print("your score is",score)
```

43.

```
# Tic-Tac-Toe - Aaron Bernath #
# --------- Global Variables ----------

# Will hold our game board data
board = ["-", "-", "-",
         "-", "-", "-",
         "-", "-", "-"]

# Lets us know if the game is over yet
game_still_going = True

# Tells us who the winner is
winner = None

# Tells us who the current player is (X goes first)
current_player = "X"


# ------------- Functions ---------------

# Play a game of tic tac toe
def play_game():

 # Show the initial game board
 display_board()

 # Loop until the game stops (winner or tie)
 while game_still_going:

  # Handle a turn
  handle_turn(current_player)

  # Check if the game is over
  check_if_game_over()

  # Flip to the other player
  flip_player()
```

```
  # Since the game is over, print the winner or tie
  if winner == "X" or winner == "O":
    print(winner + " won.")
  elif winner == None:
    print("Tie.")


# Display the game board to the screen
def display_board():
  print("\n")
  print(board[0] + " | " + board[1] + " | " + board[2] + "     1 | 2 | 3")
  print(board[3] + " | " + board[4] + " | " + board[5] + "     4 | 5 | 6")
  print(board[6] + " | " + board[7] + " | " + board[8] + "     7 | 8 | 9")
  print("\n")


# Handle a turn for an arbitrary player
def handle_turn(player):

  # Get position from player
  print(player + "'s turn.")
  position = input("Choose a position from 1-9: ")

  # Whatever the user inputs, make sure it is a valid input, and the spot is open
  valid = False
  while not valid:

    # Make sure the input is valid
    while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:
      position = input("Choose a position from 1-9: ")

    # Get correct index in our board list
    position = int(position) - 1

    # Then also make sure the spot is available on the board
    if board[position] == "-":
      valid = True
    else:
      print("You can't go there. Go again.")

  # Put the game piece on the board
  board[position] = player

  # Show the game board
  display_board()


# Check if the game is over
def check_if_game_over():
  check_for_winner()
  check_for_tie()


# Check to see if somebody has won
def check_for_winner():
  # Set global variables
  global winner
  # Check if there was a winner anywhere
  row_winner = check_rows()
  column_winner = check_columns()
  diagonal_winner = check_diagonals()
```

```
  # Get the winner
  if row_winner:
    winner = row_winner
  elif column_winner:
    winner = column_winner
  elif diagonal_winner:
    winner = diagonal_winner
  else:
    winner = None


# Check the rows for a win
def check_rows():
  # Set global variables
  global game_still_going
  # Check if any of the rows have all the same value (and is not empty)
  row_1 = board[0] == board[1] == board[2] != "-"
  row_2 = board[3] == board[4] == board[5] != "-"
  row_3 = board[6] == board[7] == board[8] != "-"
  # If any row does have a match, flag that there is a win
  if row_1 or row_2 or row_3:
    game_still_going = False
  # Return the winner
  if row_1:
    return board[0]
  elif row_2:
    return board[3]
  elif row_3:
    return board[6]
  # Or return None if there was no winner
  else:
    return None


# Check the columns for a win
def check_columns():
  # Set global variables
  global game_still_going
  # Check if any of the columns have all the same value (and is not empty)
  column_1 = board[0] == board[3] == board[6] != "-"
  column_2 = board[1] == board[4] == board[7] != "-"
  column_3 = board[2] == board[5] == board[8] != "-"
  # If any row does have a match, flag that there is a win
  if column_1 or column_2 or column_3:
    game_still_going = False
  # Return the winner
  if column_1:
    return board[0]
  elif column_2:
    return board[1]
  elif column_3:
    return board[2]
  # Or return None if there was no winner
  else:
    return None


# Check the diagonals for a win
def check_diagonals():
  # Set global variables
  global game_still_going
  # Check if any of the columns have all the same value (and is not empty)
```

```
  diagonal_1 = board[0] == board[4] == board[8] != "-"
  diagonal_2 = board[2] == board[4] == board[6] != "-"
  # If any row does have a match, flag that there is a win
  if diagonal_1 or diagonal_2:
    game_still_going = False
  # Return the winner
  if diagonal_1:
    return board[0]
  elif diagonal_2:
    return board[2]
  # Or return None if there was no winner
  else:
    return None


# Check if there is a tie
def check_for_tie():
  # Set global variables
  global game_still_going
  # If board is full
  if "-" not in board:
    game_still_going = False
    return True
  # Else there is no tie
  else:
    return False


# Flip the current player from X to O, or O to X
def flip_player():
  # Global variables we need
  global current_player
  # If the current player was X, make it O
  if current_player == "X":
    current_player = "O"
  # Or if the current player was O, make it X
  elif current_player == "O":
    current_player = "X"


# ------------ Start Execution -------------
# Play a game of tic tac toe
play_game()
```

44.

```
""" Tic-Tac-Toe players using inheritance implementation by Kylie Ying """
""" module player """
import math
import random


class Player():
    def __init__(self, letter):
        self.letter = letter

    def get_move(self, game):
        pass


class HumanPlayer(Player):
    def __init__(self, letter):
        super().__init__(letter)
```

```
    def get_move(self, game):
        valid_square = False
        val = None
        while not valid_square:
            square = input(self.letter + '\'s turn. Input move (0-9): ')
            try:
                val = int(square)
                if val not in game.available_moves():
                    raise ValueError
                valid_square = True
            except ValueError:
                print('Invalid square. Try again.')
        return val


class RandomComputerPlayer(Player):
    def __init__(self, letter):
        super().__init__(letter)

    def get_move(self, game):
        square = random.choice(game.available_moves())
        return square


class SmartComputerPlayer(Player):
    def __init__(self, letter):
        super().__init__(letter)

    def get_move(self, game):
        if len(game.available_moves()) == 9:
            square = random.choice(game.available_moves())
        else:
            square = self.minimax(game, self.letter)['position']
        return square

    def minimax(self, state, player):
        max_player = self.letter  # yourself
        other_player = 'O' if player == 'X' else 'X'

        # first we want to check if the previous move is a winner
        if state.current_winner == other_player:
            return {'position': None, 'score': 1 * (state.num_empty_squares() + 1) if other_player == max_player else -1 * (
                    state.num_empty_squares() + 1)}
        elif not state.empty_squares():
            return {'position': None, 'score': 0}

        if player == max_player:
            best = {'position': None, 'score': -math.inf}  # each score should maximize
        else:
            best = {'position': None, 'score': math.inf}  # each score should minimize
        for possible_move in state.available_moves():
            state.make_move(possible_move, player)
            sim_score = self.minimax(state, other_player)  # simulate a game after making that move

            # undo move
            state.board[possible_move] = ' '
            state.current_winner = None
            sim_score['position'] = possible_move  # this represents the move optimal next move

            if player == max_player:  # X is max player
                if sim_score['score'] > best['score']:
```

```
            best = sim_score
        else:
            if sim_score['score'] < best['score']:
                best = sim_score
    return best

""" module game """
import math
import time
from player import HumanPlayer, RandomComputerPlayer, SmartComputerPlayer


class TicTacToe():
    def __init__(self):
        self.board = self.make_board()
        self.current_winner = None

    @staticmethod
    def make_board():
        return [' ' for _ in range(9)]

    def print_board(self):
        for row in [self.board[i*3:(i+1) * 3] for i in range(3)]:
            print('| ' + ' | '.join(row) + ' |')

    @staticmethod
    def print_board_nums():
        # 0 | 1 | 2
        number_board = [[str(i) for i in range(j*3, (j+1)*3)] for j in range(3)]
        for row in number_board:
            print('| ' + ' | '.join(row) + ' |')

    def make_move(self, square, letter):
        if self.board[square] == ' ':
            self.board[square] = letter
            if self.winner(square, letter):
                self.current_winner = letter
            return True
        return False

    def winner(self, square, letter):
        # check the row
        row_ind = math.floor(square / 3)
        row = self.board[row_ind*3:(row_ind+1)*3]
        # print('row', row)
        if all([s == letter for s in row]):
            return True
        col_ind = square % 3
        column = [self.board[col_ind+i*3] for i in range(3)]
        # print('col', column)
        if all([s == letter for s in column]):
            return True
        if square % 2 == 0:
            diagonal1 = [self.board[i] for i in [0, 4, 8]]
            # print('diag1', diagonal1)
            if all([s == letter for s in diagonal1]):
                return True
            diagonal2 = [self.board[i] for i in [2, 4, 6]]
            # print('diag2', diagonal2)
            if all([s == letter for s in diagonal2]):
                return True
        return False
```

```
    def empty_squares(self):
        return ' ' in self.board

    def num_empty_squares(self):
        return self.board.count(' ')

    def available_moves(self):
        return [i for i, x in enumerate(self.board) if x == " "]


def play(game, x_player, o_player, print_game=True):

    if print_game:
        game.print_board_nums()

    letter = 'X'
    while game.empty_squares():
        if letter == 'O':
            square = o_player.get_move(game)
        else:
            square = x_player.get_move(game)
        if game.make_move(square, letter):

            if print_game:
                print(letter + ' makes a move to square {}'.format(square))
                game.print_board()
                print('')

            if game.current_winner:
                if print_game:
                    print(letter + ' wins!')
                return letter  # ends the loop and exits the game
            letter = 'O' if letter == 'X' else 'X'  # switches player

        time.sleep(.8)

    if print_game:
        print('It\'s a tie!')



if __name__ == '__main__':
    x_player = SmartComputerPlayer('X')
    o_player = HumanPlayer('O')
    t = TicTacToe()
    play(t, x_player, o_player, print_game=True)
```

45.

```
from tkinter import *
from time import *

def update():
    time_string = strftime("%I:%M:%S %p")
    time_label.config(text=time_string)

    day_string = strftime("%A")
    day_label.config(text=day_string)

    date_string = strftime("%B %d, %Y")
    date_label.config(text=date_string)
```

```
    window.after(1000,update)


window = Tk()

time_label = Label(window,font=("Arial",50),fg="#00FF00",bg="black")
time_label.pack()

day_label = Label(window,font=("Ink Free",25,"bold"))
day_label.pack()

date_label = Label(window,font=("Ink Free",30))
date_label.pack()

update()

window.mainloop()
```

46.

```
import random

while True:
    choices = ["rock","paper","scissors"]

    computer = random.choice(choices)
    player = None

    while player not in choices:
        player = input("rock, paper, or scissors?: ").lower()

    if player == computer:
        print("computer: ",computer)
        print("player: ",player)
        print("Tie!")

    elif player == "rock":
        if computer == "paper":
            print("computer: ", computer)
            print("player: ", player)
            print("You lose!")
        if computer == "scissors":
            print("computer: ", computer)
            print("player: ", player)
            print("You win!")

    elif player == "scissors":
        if computer == "rock":
            print("computer: ", computer)
            print("player: ", player)
            print("You lose!")
        if computer == "paper":
            print("computer: ", computer)
            print("player: ", player)
            print("You win!")

    elif player == "paper":
        if computer == "scissors":
            print("computer: ", computer)
            print("player: ", player)
            print("You lose!")
        if computer == "rock":
            print("computer: ", computer)
```

```
            print("player: ", player)
            print("You win!")

        play_again = input("Play again? (yes/no): ").lower()

        if play_again != "yes":
            break

    print("Bye!")
```

47.

```
    import random

    def play():
        user = input("What's your choice? 'r' for rock, 'p' for paper, 's' for scissors\n")
        computer = random.choice(['r', 'p', 's'])

        if user == computer:
            return 'It\'s a tie'

        # r > s, s > p, p > r
        if is_win(user, computer):
            return 'You won!'

        return 'You lost!'

    def is_win(player, opponent):
        # return true if player wins
        # r > s, s > p, p > r
        if (player == 'r' and opponent == 's') or (player == 's' and opponent == 'p') \
            or (player == 'p' and opponent == 'r'):
            return True

    print(play())
```

48.

```
    # # string concatenation (aka how to put strings together)
    # # suppose we want to create a string that says "subscribe to _____ "
    # youtuber = "Kylie Ying" # some string variable

    # # a few ways to do this
    # print("subscribe to " + youtuber)
    # print("subscribe to {}".format(youtuber))
    # print(f"subscribe to {youtuber}")

    adj = input("Adjective: ")
    verb1 = input("Verb: ")
    verb2 = input("Verb: ")
    famous_person = input("Famous person: ")

    madlib = f"Computer programming is so {adj}! It makes me so excited all the time because \
    I love to {verb1}. Stay hydrated and {verb2} like you are {famous_person}!"

    print(madlib)
```

49.

```
    import random

    def guess(x):
        random_number = random.randint(1, x)
        guess = 0
        while guess != random_number:
```

```
        guess = int(input(f'Guess a number between 1 and {x}: '))
        if guess < random_number:
            print('Sorry, guess again. Too low.')
        elif guess > random_number:
            print('Sorry, guess again. Too high.')

    print(f'Yay, congrats. You have guessed the number {random_number} correctly!!')

def computer_guess(x):
    low = 1
    high = x
    feedback = ''
    while feedback != 'c':
        if low != high:
            guess = random.randint(low, high)
        else:
            guess = low  # could also be high b/c low = high
        feedback = input(f'Is {guess} too high (H), too low (L), or correct (C)?? ').lower()
        if feedback == 'h':
            high = guess - 1
        elif feedback == 'l':
            low = guess + 1

    print(f'Yay! The computer guessed your number, {guess}, correctly!')


guess(10)
```

50.

```
""" module words """
words = ["aback","abaft","abandoned"]

""" hangman_visual """
lives_visual_dict = {
        0: """
            ___________
            | /       |
            |/       ( )
            |         |
            |        / \\
            |
          """,
        1: """
            ___________
            | /       |
            |/       ( )
            |         |
            |        /
            |
          """,
        2: """
            ___________
            | /       |
            |/       ( )
            |         |
            |
            |
          """,
        3: """
            ___________
            | /       |
            |/       ( )
```

```
            |
            |
            |
        """,
    4: """
            _________
            | /     |
            |/
            |
            |
            |
        """,
    5: """
            _________
            | /
            |/
            |
            |
            |
        """,
    6: """
            |
            |
            |
            |
            |
        """,
    7: "",
}

""" hangman_visual """
import random
from words import words
from hangman_visual import lives_visual_dict
import string


def get_valid_word(words):
    word = random.choice(words)  # randomly chooses something from the list
    while '-' in word or ' ' in word:
        word = random.choice(words)

    return word.upper()


def hangman():
    word = get_valid_word(words)
    word_letters = set(word)  # letters in the word
    alphabet = set(string.ascii_uppercase)
    used_letters = set()  # what the user has guessed

    lives = 7

    # getting user input
    while len(word_letters) > 0 and lives > 0:
        # letters used
        # ' '.join(['a', 'b', 'cd']) --> 'a b cd'
        print('You have', lives, 'lives left and you have used these letters: ', ' '.join(used_letters))

        # what current word is (ie W - R D)
        word_list = [letter if letter in used_letters else '-' for letter in word]
        print(lives_visual_dict[lives])
```

```
            print('Current word: ', ' '.join(word_list))

            user_letter = input('Guess a letter: ').upper()
            if user_letter in alphabet - used_letters:
                used_letters.add(user_letter)
                if user_letter in word_letters:
                    word_letters.remove(user_letter)
                    print('')

                else:
                    lives = lives - 1  # takes away a life if wrong
                    print('\nYour letter,', user_letter, 'is not in the word.')

            elif user_letter in used_letters:
                print('\nYou have already used that letter. Guess another letter.')

            else:
                print('\nThat is not a valid letter.')

        # gets here when len(word_letters) == 0 OR when lives == 0
        if lives == 0:
            print(lives_visual_dict[lives])
            print('You died, sorry. The word was', word)
        else:
            print('YAY! You guessed the word', word, '!!')


    if __name__ == '__main__':
        hangman()
```

51.

```
    class Car: # it is called the constructor
      def__init__(self,make,model,year,color): # this is the constructor that we need it or it will not work
        self.make=make #instance variable
        self.model=model #instance variable
        self.year=yea #instance variable
        self.color=color #instance variable
      def drive(self): # we need 1 argument; self refers to the object that uses this method
        print("This car is driving")
      def stop(self):
        print("This car is stopped")

    car_1 = Car("Chevy","Corvette",2021,"Blue")
    car_2 = Car("For","Mustang",2022,"red")
    print(car_1.make)
    print(car_1.model)
    print(car_1.year)
    print(car_1.color)
    car_1.drive()
    car_1.stop()
    car_2.drive()
    car_2.stop()

    # you can write
    def drive(self):
    print("This " + self.model+" car is driving")
```

52.

```
    use class variable
    from car import Car

    car_1 = Car("Chevy","Corvette",2021,"Blue")
```

```
car_2 = Car("For","Mustang",2022,"red")

class Car:
  wheels = 4 # class variable - created within the class but outside the constructor - you can set a default value for all instances of this class for all unique values created and can be changed later if needed
def__init__(self,make,model,year,color):
  self.make=make #instance variable - declared inside a constructor and given unique values
  self.model=model #instance variable - declared inside a constructor and given unique values
  self.year=year #instance variable - declared inside a constructor and given unique values
  self.color=color #instance variable - declared inside a constructor and given unique values

car_1.wheels = 2

print(car_1.wheels) # will print 2
print(car_2.wheels) # will print 4

Car.wheels = 2
Then it will print all instances as 2

or you can do it like this:
print(Car.wheels)
```

53.

```
use Inheritance
class Animal:
  alive = True
  def eat(self):
    print("This animal is eating")
  def sleep(self):
    print("This animal is sleeping")

class Rabbit(Animal): # so rabbit is the child class and animal is the parent class, # use pass if you want to add later info for the class
  def run(self):
    print("This rabbit is running")

class Fish(Animal):
  def swim(self):
    print("This fish is swimming")

class Hawk(Animal):
  def fly(self):
    print("This hawk is flying")

rabbit=Rabit() # create object from this classes, # each of those children classes will inherit from parent class
fish=Fish()
hawk=Hawk()
print(rabbit.alive)
fish.eat()
hawk.sleep()
rabbit.run()
fish.swim()
hawk.fly()
```

54.

```
use Multilevel Inheritance
class Organism:
  alive = True # attribute of alive

class Animal(Organism):
  def eat(self):
    print("This animal is eating")

class Dog(Animal):
```

```
  def bark(self):
    print("This dog is barking")

dog=Dog() # create an object
print(dog.alive)
dog.eat()
dog.bark()
```

55.

use Multiple Inheritance

```
class Prey:
  def flee(self):
    print("This animal flees")

class Predator:
  def hunt(self):
    print("This animal is hunting")

class Rabbit(Prey):
  pass

class Hawk(Predator):
  pass

class Fish(Prey, Predator)
  pass

rabbit = Rabbit()
hawk = Hawk()
fish = Fish()
rabbit.flee()
hawk.hunt()
fish.flee()
fish.hunt(
```

56.

use Method Overriding

```
class Animal:
  def eat(self):
    print("This animal is eating")

class Rabbit(Animal):
  def eat(self):
    print("This rabbit is eating a carrot!") # so, this will override the main method from parent class

rabbit=Rabbit()
rabbit.eat()
```

57.

Use Method Chaining

```
class Car:
  def turn_on(self):
    print("You start the engine")
   return self # you must add this for Method Chaining

  def drive(self):
    print("You drive the car")
  return self

  def brake(self):
    print("You step on the brakes")
  return self
```

```
 def turn_off(self):
   print("You turn off the engine")
  return self

car=Car()
car.turn_on().drive().
car.brake().turn_off()
car.turn_on\
.drive()\ # \ is line continuation
.brake()\
.turn_off()
```

58.

```
import datetime

class User:
   """ A member of FriendFace. For now we are
   only storing their name and birthday.
   But soon we will store an uncomfortable
   amount of user information"""

   def __init__(self, full_name, birthday):
      self.name = full_name
      self.birthday = birthday # yyymmdd

      #extract first and last name
      name_pieces = full_name.split(" ")
      self.first_name = name_pieces[0]
      self.last_name = name_pieces[-1]

   def age(self):
      """Return the age of the user in years""" # <- Brief Docstring about what below will do
      today = datetime.date(2001, 5, 12)
      # we will convert the birthday string into a date object
      yyyy = int(self.birthday[0:4]) # extract year as int
      mm = int(self.birthday[4:6]) # extract month
      dd = int(self.birthday[6:8]) # extract the day
      dob = datetime.date(yyyy, mm, dd) # Date of birth
      age_in_days = (today - dob).days # by computing the difference ewe will get time delta object that will have a field called days
      age_in_years = age_in_days / 365
      return int(age_in_years)

user = User("Dave Bowman","19710315")
print(user.age()) # you did not type self when calling the age method as the self keyword is only used when writing the method

help(User)
```

- variations of starting variables (9):
  - keepGoing = True || Boolean expression
  - name = None || None is just a value that commonly is used to signify 'empty', or 'no value here'. It is a signal object; it only has meaning because the Python documentation says it has that meaning, so it means absence of a value
  - feedback = ' ' || Empty String
  - correct = "python" || String
  - tries = 3 || Integer
  - guesses = [ ] || Empty list
  - choices = ["rock","paper","scissors"] || List With Items
  - largest_number = numbers[0] || Index
  - computer = random.choice(['r','p','s']) || Chosen Random Letters (import random)

- variations of if/elif/else (8):
  1. if/elif/else with 'in', 'and', 'or', 'not in': use in when it comes to str (word="lulu" -> if "l" or "l" in word)
  2. if/elif/else with Boolean expression (var,input,def): employed
  3. if/elif/else with condition: earn 200$
  4. use nested if's: variations where to go based on money in the pocket -> if money <=200: -> print("") -> else: -> print("you will go to.....shopping mall")
  5. if with len() [username]: username=input("") -> if 4 <= len(username) <= 10: -> print(f"thank you {username} ok") -> else: -> print("this is wrong input")
  6. if with replace (dash): name="aloha" -> if "a" in word: -> new_word=name.replace("a","_") -> print(name)
  7. if with indexing: if (name[0].islower()): -> name = name.capitalize()
     -> index operator [] = gives access to a sequence's element (str, list, tuples)
  8. if with "is" and or "is not" (best example of mutable vs immutable): compare 2 strings OR 2 tuples
     -> compare 2 variables (don't use input from user just direct values in the variables) => the id's must match
     ->" is" and "is not" will compare 2 variables that are stored in same place when we add direct value in a variable, but if we use input then the variable passed through as string will be allocated in memory in a different place
     -> so using "is" or "is not" with list, dictionary, user input, etc (mutable) will NOT WORK -> better use == or != if you take any input or list, dictionary etc. (mutable)
     -> str, integers, float and tuples are immutable -> they will have same id's in the memory -> it WILL WORK (tuple must have same value inside if you will compare it with other tuple)

- variations of WHILE loop (break,continue,pass,join) ➢ use for var with ONLY 1 data (5):
  1. while with if, "in" and "or": letter word," ","-"
  2. while with expression: guess word & lives -> while keepGoing, while tripFinished !="YES", while feedback !='c'
  3. while with comparison operator: curry spoons -> while <=5
  4. while with len(): username input instead of blank space -> while len(variable)
  5. use multiple while: days of the week
  6. while with "not" and "not in" using a Boolean variable and a list: check if input already chosen or not by user and that it does not overlap
  7. while with "else"
  8. while with "True" at the beginning and use break to end the loop at the end
     1. while with "in" and "or": word="lulu" -> keep_going=True -> while keep_going: -> if "l" or "u" in word: -> print("the letters l and u are in the word") -> keep_going=False
     2. while with expression:
        - keepGoing=True -> correct="python" -> tries=3 -> while keepGoing: -> guess=input("") -> tries=tries-1 -> if guess == correct: -> print("") -> keepGoing=False -> else: -> print("")
          -> that expression we need to define it before we enter the while loop, the var should be initialized so we can tell python it exists, only then we can use it
          -> make sure if there is a number in the var before while then you have to use int otherwise if you press a number from the keyboard as input it won't work
          -> don't forget to close the Boolean var
        - tripFinished="NO" -> while tripFinished !="YES": || so anything but yes then loop
        - low = 1 -> high=10 -> feedback=' ' -> while feedback !='c': <- so you use feedback var for 2 things - to start the loop and also to compare to c to end the loop when needed
     3. while with condition:
        1. take a counter
        2. choose while
        3. specify a condition
        4. check if the value of counter is less than a number
        5. use ":" which means that what is coming ahead is a part of this block and following statements belong to the same suit
        6. print statement
        7. print value of counter

* 3 key points questions while loop has to answer:

3.1. initialization: spoon=1 (how does my sentry(spoon) start) <- start by taking 1 spoon of curry

3.2. condition: spoon <=5 (when will my sentry(spoon) end) <- did I eat 5 spoons of curry? if yes then stop the loop

-> print ("you ate",spoon,"spoons of curry") <- advise me how many spoons of curry I had up till now

3.3. increment/decrement: spoon=spoon+1 (how does my sentry(spoon) change) <- increase so I will take another spoon of curry <- without i=i+1 the loop will be forever

4. while with len():

```
""" insert username input
instead of blank space """

name = ""
while name == "":
  name = input("there's nothing here, please insert a word: ")
print("hello" + name)
```

5. use multiple while (days of the week): like using days of the week -> each day has 24 hours -> it loops through those hours in the day like a clock and when finished it jumps to the next day and does the same thing again

-> so, the outer loop sentry will represent the number days and the inner loop sentry will represent the hours

-> print the day 1 time then 4 times hours -> then repeat for 4 times same thing

```
i=1
while i <=5:
 print("Day ",end="")
 j=1
 while j<=4:
   print("hours ",end="")
   j=j+1
 i=i+1
 print()
```

debug:

i value is 1

i isn't 5: yes

it will print **Day** and not go on new line, it will **STAY** on same line

the value for j is 1

is value of j less than 4: yes

it will go ahead and print **hours** on the same line

value of j will be incremented

value of j will be 2

when the value of j will be 5 which is more than 4 it will come out to i=i+1

so it will increment i

do all the above again

output:

```
Day hours hours hours hours
Day hours hours hours hours
Day hours hours hours hours
Day hours hours hours hours
```

6. while with "not" and "not in" using a Boolean variable and a list: check if input already chosen or not by user and that it does not overlap

```
board = ["-","-","-","-","-","-","-","-","-","-","-","-","-","-","-","-"]
valid = False <- Boolean variable
position = None <- empty variable to be filled with user input

while not valid:
 while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]: <- if my next input which will be a string is not in this list then loop
  position = input("please input a number from 1 to 9: ")

  position = int(position) - 1 # <- cast the input

  if board[position] == "-":
   valid = False # <- if true and there will be no "-" then it will break from the Boolean value
  else:
   print()
   print("you can't go there, try again")
   print()
 board[position] = "x"
 print(board)
```

7. while with "else":

```
secret_number = 9
guess_count = 0
guess_limit = 3

while guess_count < guess_limit:
 guess = int(input('Guess: '))
 guess_count += 1 # <-you can add to reduce the guess limit
 if guess == secret_number:
  print("You have won!")
  break
else:
print("sorry you have failed")
```

8. while with "True" at the beginning and use break to end the loop at the end: **while True:.....elif command == "quit":break -> else:->print("sorry I don't understand")**

- variations of FOR loop (break,continue,pass,join) ➢ use for var with MORE than 1 data (7):
  1. for one under each other from a list: **menu below each other**
  2. for one next to each other in a list:
     - 2.1.1. expression for item in iterable:
       - a. for str menu next to each other
       - b. for numbers: count 2 trees at a time
     - 2.1.2. expression if/else for item in iterable: **compare 2 lists and change letter to "_"**
     - 2.1.3. expression for item in iterable if conditional:
       - a. print a list of passing marks
       - b. print a list of passing marks and the rest failed
  3. for with direct tuple or list next to it: **for i in (2,6,'Paul'):-> print(i) // for i in [2,6,3]:-> print(i)**
  4. for loop with index OR item:
     - a. index: print a range of students in a class of 20
     - b. item: check specific fruit in basket and print that fruit if found
  5. for with var at index: **highest mark from marks**
  6. iterate with for loop through dictionary: **capitals of countries**
  7. use nested for: **print symbol over rows and columns**
  8. for with var set to 0
  9. iterate with for into a 2D list

1. for one under each other from a list:

word=["lulu",65,2.5] -> for letter in word: -> print(letter)

-> unlike while,we don't initialize the var, we don't take a condition (if/elif/else), and we don't increment or decrement (will be done by for loop)-> we use a variable (that will represent one element of the list at a time) to go through the list

-> for **WHILE** loop, you setup a **Condition** or **Expression** but for the **FOR** loop it normally works with **Iterating Sequence of (list,tuple,string) or a string**

-> you can iterate in: **[]** or **()** or **{}** or something in **var without parenthesis**

2. for one next to each other in a list:
   - 2.1. use comprehension list (for item,if/else for item,for item in iterable if conditional):
     - 2.1.1. variation: **list = [expression for item in iterable]**

-> with string: list=[letter for letter in word] -> print(list)

-> with numbers: squares = [i * i for i in range(1,11)] -> print(squares) // instead of writing i in range(1,11)

board = [' ' for _ in range(9)] -> print(board) // instead of writing board = [' ',' ',' ',' ',' ',' ',' ',' ',' ']

board = [' ' for _ in range(9)] -> for row in [board[i*3]] for i in range(3)]: -> print('|' + '|'.join(row)+'|') // of writing: print(board[0] + " | " + board[1] + " | " + board[2]) …

     - 2.1.2. variation: list = [expression if/else for item in iterable] **-> word=("lulu") -> one=("u") -> list -> list[letter if letter in one else "_" for letter in word] -> print(list)**
     - 2.1.3. variation: list = [expression for item in iterable if conditional]

-> students = [100,90,80,70,60,50,40,30,0] -> passed_students=[i for i in students if i>=60]

-> students = [100,90,80,70,60,50,40,30,0] -> passed_students=[i if i>=60 else "Failed" for i in students] -> print(passed_students)

3. for with direct tuple or list next to it: **for i in (2,6,'Paul'):-> print(i) // for i in [2,6,3]:-> print(i)**
4. for loop with index OR item:
   - 4.1. iterate by index: for x in range(0,10,1): **-> #start,stop,step**
     - 4.1.1. variation to iterate by index: for i in range (10): **->** print(i) **//** range(1,11) or range(0,30,5)

-> i is the sentry variable (what's my sentry, how does it start, how does it end, how does it change)

   - 4.2. iterate by item: for fruit in fruits: **->** if fruit=='pears' **->** print(fruit)
     - 4.2.1. variation to iterate by item: for x in range(0,6): **->** if fruits[x]=='pears' **->** print(fruit[x])
     - 4.2.2. variation to iterate by item: for x in range(len(fruits)): **->** if fruits[x]=='pears' **->** print(fruit[x])
5. for with var at index: heights = [100,2,300,10,11,1000] **->** largest_number = heights[0] **->** for number in heights: **->** if number **>** largest_number: **->** largest _number = number **->** print(largest_number) **|| 1000**
6. iterate with for loop through dictionary**:** you will not know in what order they will come **->** for state,cap in stateCap.items(): **->** print ("{:15} {:15}".format(state,cap))
7. use nested for's:

```
rows = int(input("How many rows?: ")) // for outer loop
columns = int(input("How many columns?: ")) // for inner loop
symbol = input("Enter a symbol to use: ")

for i in range(rows):
  for j in range(columns):
     print(symbol, end="")
  print()
```

8. for with var set to 0:

```
prices = [10,20,30]
total = 0
for price in prices:
  total += price
print(total)
```

9. iterate with for into a 2D list:

```
matrix = [
[1,2,3],
[4,5,6],
[7,8,9]
]
for row in matrix:
  for item in row:
    print(item)
```

- create empty set, list, string, dictionary & tuple:
  - you can create can empty set() and tuple() with the word 'set' and 'tuple' next to () but you cannot create list[] with the word 'list' next to it just [] or dictionary{} with the word 'dictionary' next to it just {}
  - create empty set: used_letters = set( )
    -> create set by using a list: will separate word into letters -> words=["lulu"] -> word_letters=set(words) -> word_letters.remove(user_letter)
    -> create set of alphabet letters: import string -> alphabet=set(string.ascii_uppercase)
  - create empty string: empty=" " -> user=input('something') -> empty=user -> print(user)
  - create empty dictionary: test = { } -> add to dictionary: test.update({"lulu":100}) -> print(test)
    -> there is no add, append or insert method in dictionary -> add with update ({:}) and remove with pop('')
    -> iterate through a dictionary with for key,value in capitals.items():- > print(key,value)
    -> print from dictionary: print(capitals['germany']) // get value -> print(capitals.get('john'))
    -> ic.update({'':''}),dic.keys(),dic.values(),dic.items(),clear(),.dic.pop('')
  - create empty list: test = [ ] -> test.append("lulu") -> print(test)
    -> initialize to an empty list: def moves(self): -> moves[ ] <- create empty list using name of def
  - you CANNOT create in tuple -> variable.count(''), variable.index('') <- tuple has only those methods, so no adding methods etc.

- conversions:
  - convert and use .join in the print for any iterable object lists and sets -> print('you have chosen: ', ' '.join(used_letters)
  - convert 1 string: word='lulu' -> list: letter=list(word) -> to set: letter=set(word) -> to tuple: letter=tuple(word) -> to dictionary: letter=dict.fromkeys(word)
    -> ordered list = tuple, list, dictionary
    -> un-ordered list = set
    -> list will show all letters form a word
    -> set() will take all duplicate letters and remove them (same goes for dictionary)
    * list tools [ ]: variable.remove(''), variable.sort(reverse=true/false), variable.pop('2'), variable.clear(),variable.insert(index,item), variable.append('') ||pop removes from index,remove removes item
    * set() tools { }: variable.clear(),variable.union(set), variable.intersection(set), variable.difference(set), variable.remove(item), variable.add(item), variable.update(set)
    * tuple() tools( ): variable.count(''), variable.index('')
    * dict.fromkeys() tools { }: dic.update({'':''}) -> letter.update({'k":100}) -> print(letter)
    -> variable.update({'':''}), variable.keys(),variable.values(),variable.items(),clear(),.variable.pop('') <- but like set it will not print double values and it will be something like 'the letter':None
    -> adding element to a dictionary with array-like syntax: stateCap["Florida"]="Tallahassee"
  - convert list with more than 1 string: if you have a list with MORE than one string -> ex. words=['lulu','aloha'] -> take out a word out first by using random -> word=random.chice(words) -> letter=set(word)
    -> word = set(words[0])

- variations of list (1):

1. list index replacement using input:

```
flavour = ["banana", "chocolate"]
new_flavour = input("please insert new flavour: ")
flavour[0] = new_flavour
```

```
print(flavour)
```

- variations of def (11):
  1. FEED arguments from the caller to def parameters so that the def will DO tasks for you (hey guys, call the manager and say "tomorrow I will be late for the meeting tomorrow")
  2. create functions to MAKE something (hey guys, call the manager and "ask the manager what time is the meeting tomorrow")
  3. use keyword to pass arguments from caller to the parameter you want
  4. change index of a list inside a def
  5. use * to pass how many arguments you want (where a will be 5 and *b will be the rest as a tuple)
  6. use default if you don't want to pass all arguments from the caller
  7. call function from another function
  8. pass from list from another module
  9. put caller in a var and use it later
  10. pass variable or list as argument through a function
  11. mix of function with global variable, Boolean variable of other function and return
  12. use function arguments in other function parameters

1. FEED arguments from the caller to def parameters so that the def will DO tasks for you (hey guys, call the manager and say "tomorrow I will be late for the meeting tomorrow"):
   - feed the caller's argument => caller argument will be transformed into def parameters then transformed into def variable that can be used anywhere in def

a. generate a random number from 1 to a number given by user, and repeat that request for the user:

```
""" generate random number from 1
to a number given by user """

import random

def user_random_number(x):
  random_number = random.randint(1,x)
  print("Your random number entered is:{}".format(x), "and the random number is:",random_number)

user_random_number(int(input("Please enter ending number: ")))
```

b. add 2 numbers given by the user, and repeat that request for the user:

```
""" add 2 numbers
given by the user """

keep_going = True

def add_numbers(x,y):
  x = int(input("please enter x: "))
  y = int(input("please enter y: "))
  sum = x + y
  print(sum)

while keep_going:
  add_numbers(1,1)
```

2. create functions to MAKE something by feeding the caller same as in 1 an argument but this time we save the result in a value (hey guys, "ask the manager what time is the meeting tomorrow"):

2.1. save value instead of printing it: ant march using global and local variables by asking user for the ant number with a multi-line string return output

-> you can put data in var and give it a name then return it c="hello" return c => you can modify it: return c.upper

* Example:

```
def verse(verseNum):  and inside you can use what you fed (verseNum) as its own variable
 if verseNum == 1: #so verseNum is now a variable and I'm gonna do stuff with it
    distraction = "sing"
 elif verseNum ==2:
    distraction ="dance" # use local variable inside your def
 else:
    distraction="sit"
  output =""" # <- will be under else
    The ants are marching {0} by {0} hurrah, hurrah.
    The ants are marching {0} by {0} hurrah, hurrah.
    The ants are marching {0} by {0} hurrah, hurrah.
    The little one stops to {1} # {1} meaning distraction
    """.format(verseNum, distraction)
  return output # <- will be under else  -> return output (so last line of the function is the statement return) || print(verse(2)) or print(verse(2)) (because now the function does not print the verse 2 or 1, it creates the verse 1 or 2)
print(verse(2))                          -> so after you make function create things you can have flexibility, ex.take chorus and write it to a file, send it to a text to speech engine, I want to append it to a web page
```

print(verse(1))

2.2. return 2 values in a function: add and subtract same numbers using same function -> def add_sub(x,y) -> c=x+y -> d=x-y -> return c,d -> result1,result2 =add_sub(5,4) [make sure you assign to a new var in order to print]

3. use keyword to pass arguments from caller to the parameter you want:

-> person(age=29,name="lulu") -> def person(name,age)

4. change index of a list inside a def:

```
board = ["-","-","-",
         "-","-","-",
         "-","-","-"] <=== ["-","-","-","-","-","-","-","-","-","-","-","-"]

def display_board():
  print(board[0] + "|" + board[1] + "|" + board[2]) <===> -|-|-

display_board()
```

5. use * to pass how many arguments you want (where a will be 5 and *b will be the rest as a tuple):

```
""" create function
to add any number """

def add (*numbers): <- because you cannot add int with tuple so you have to iterate through it
  sum = 0

  for i in numbers:
    sum+ = i
  print(sum)
                -> add(1,1,5,5,20)
```

6. use default if you don't want to pass all arguments from the caller: person('lulu') -> def person(name, age=18)

7. call function from another function: def check_if_game_over(): -> check_if_win() -> check_if_tie() -> return

8. pass from list from another module: use import to inject into def -> import -> from words import words -> with import you can use def to bring it: def get_valid_word(words)

9. put caller in a var and use it later: testing = test() -> print(testing)

10. pass variable or list as argument through a function:

```
container_name1 = "lulu"
container_name2 = ["lulu","aloha"]

def global_argument(container1):
  print(container1)

def global_argument1(container1):
  print(container1)

global_argument(container_name1) <== "lulu"
global_argument1(container_name2[1]) <=== "aloha"
```

11. mix of function with global variable, Boolean variable of other function and return:

```
#global variables#
board=["-","-","-"]
winner = None
game_still_going = True

#main def engine#
def check_for_winner():
  global winner # <- use a global variable (usually you can only read a global variable but by using the word "global" you can overrite the global variable)

  row_winner = check_rows() # <- alocate a function to a boolean variable inside a function

  if row_winner: <- use the boolean variable from a function as a variable => acts as keep_going
    winner = row_winner # alocate the boolean variable of the function to a global variable
  else:
    winner = None <- if you don't know what the var will hold in the future you can use "None" keyword, example: winner = None (could be "x" or "o")


#def checking rows and returns it used for main def engine#
def check_rows():
  global game_still_going
```

```
row_1 = board[0] == board[1] == board[2] != "-" # <- check if any of the rows have all same value (and not empty)

if row_1: # <- if any row has a match then flag that there is a win
 game_still_going = False

if row_1:
 return board[0] # <- return the winner "x" or "o"
```

12. use function arguments in other function parameters:

```
import random
def play()
  user = input("insert r,p,s: ")
  computer = random.choice(["r","p","s"])
  if is_win(user, computer):
    return "You have won"
  if user == computer:
    return "it's a tie"
  return 'you lost'
# r > s, s>p, p > r
def is_win(player, opponent):
  # return True if player wins
  if(player == "r" and opponent == "s") or (player =="s" and opponent == "p") or (player =="p" and opponent == "r"):
    return True
print(play())
```

- variations of class (5):
  1. creating a simple class
  2. create a constructor
  3. access methods in a class using getters and setters
  4. create an inheritance
  5. overload a method

1. create a simple class:

Example 1. """ creating a simple class """

```
class Point:
  def move(self): # methods defined in the body of the class
    print("move")

  def draw(self):
    print("draw")

point1 = Point()
point1.x = 10 # set attributes from anywhere in the program
point1.y = 20
print(point1.x)
print(point1.y)
point1.draw()

point2 = Point()
point2.x = 1
print(point2.x)
```

Example 2. """ basic critter OOP """

```
class Critter(object): <- this line says I'm going to create a new class and call it Critter
  name = "Anonymous"

  def sayHi(self):
    print("Hi, my name is {}".format(self.name))

c = Critter()
print(c) <- will just print the address where it lives in the memory
c.name = "Martha"
```

```
print(c.name) # output: Martha
c.sayHi() # output: Hi my name is Martha
```

recipe (how to make the cookie) vs cookie
cookie = c = instance
↥ the recipe = Critter() = class = the class definition ↥

2. create a constructor:

Example 1. """ create a constructor """

```
class Point:
  def __init__(self,x ,y): # <- use this constructor when you create a new point object, this is to create a new object
    self.x = x # when we create a new point object, self-reference that object from memory, we use "self" to reference the current object -> we set the x ".x" attribute to the x "=x" argument passed to this function
    self.y = y

  def move(self):
    print("move")

  def draw(self):
    print("draw")

point = Point(10,20) # <- in order to use this you will need a constructor
point.x = 11
print(point.x)
```

Example 2. """ basic critter class includes a constructor """

```
class Critter(object):
  def __init__(self): <- will be automatically called as soon as we create an instance of Critter = constructor or initializer like having the lights on the board of the car when switching the key
                         the initializer will set up and give an initial value to all the attributes
    object.__init(self) <- initialize the parent
    self.name = "Anonymous" <- attribute, by using self the entire class can use it so the other functions of the object will be able to use this = same benefits as global without risks

  def sayHi(self):
    print("Hi, my name is {}!".format(self.name))

def main():
  c = Critter()
  c.name = "George"
  c.sayHi()

if __name__ == "__main__": <- so run this only if I'm on the main file, otherwise if imported, the importing file will run everything inside (because this is how python works, it runs def main, main(), so all)
  main()
```

3. access methods in a class using getters and setters:

```
""" access methods,
demonstrates getters and setters """

class Critter(object):
  def __init__(self, name = "Anonymous"):
    self.name = name

  def getName(self)
    return.name

  def setName(self, name):
    self.name = name

def main():
  c = Critter()
  c.setName("Marge")
  print(c.getName())

if __name__ == " __main__ ":
  main()
```

4. create an inheritance:

```
""" inheritance - used for reusing the code """
class Mamal:
  def walk(self):
    print("walk")

class Dog(Mamal):
  def bark(self):
    print("bark")

class Cat(Mamal):
  def be_annoying(self):
    print("annoying")

dog1 = Dog()
dog1.walk()
dog1.bark()

cat1 = Cat()
cat1.be_annoying()
```

↥ use pass if there is nothing in the method so python will not give error

5. overload a method:

```
""" overload method
(Particularly constructor) """
class Critter(object):
  def __init__(self, name ="Anonymous"): <-special parameter as it has a name but also a value
   object.__init(self) <- initialize the parent
   self.name = name <- so we are saying take the name parameter and copy it to the attribute self.name <- d.name -> "George" will be copied to name
  def sayHi(self):
    print("Hi, my name is {}!".format(self.name))

c = Critter()
c.sayHi()
d.name = ("George")
```

** TOOLBELT (ALGORITHMIC TOOLBOX) **

- **define your variables (data):** **50% of job is done,The first thing you do is think about your data**
  -> make use of variables (Boolean, normal var,assign to an emty list, make a list, set a number, set a name) before def or loops
- **type(inventory) || list // help(list)**
- **cast as integer:** int(input(f'guess a number between 1 and {x}: '))

- **to duplicate a string:** course = 'Python for Beginners' **->** another = course[:]
- **f string:** youtube=input("favorites: ") **->** print(f"subscribe to {youtube}")
  print(f'{first} [{last}] is a coder') **-> so with {} you define placeholders (creating holes in our string then you fill those holes with the value of our variables)**
- **string interpolation:** print("This is a %s %s from %s, it is %s and has %s kms." %(self.make,self.model,self.year,self.condition,self.kms)
- **string formatting:**
  - a=5 **->** b=6 **->** sum=a+b **->** print("{} + {} ={}".format (a,b,sum))
  - print("{1} is {0} years old".format(name,age)) **#reverse order**
  - print("{} is {:b} years old in binary".format(name,age))
  - print("{} is {:o} years old in octal".format(name,age))
  - print("{1:x} is {0} years old in hexodecimal".format(name,age))
  - print("{} is {:f} years old".format(name,age))
  - print("{} is {:2f} years old".format(name,age))
  - print("| {:20} |".format(name)) **#set width**
  - print("| {:<20} |".format(name)) **#justify or >20 or ^20**
  - print("The {0} jumped over the {1}".format(animal,item")) **# positional argument**
  - print("The {animal} jumped over the {item}".format(animal="cow",item="moon")) **# keyword argument**

- **print empty line:** print()
- **print multi-line strings**: print(""" text """)

- **join it:** print('Current word: ', ' '.join(list))
- **to iterate and put next to each other:** use end=" ": for letter in word: **->** prin(letter,end="")
- **loop reductor**: **->** list comprehension: return[i for i, spot in enumerate(self.board)if spot==' '] **<- condensing entire for loop in a single one line**
  -> u=[i if i in used_letters else '_' for i in word]
  -> u=[' ' for _ in range(9)]
  -> for row in [test[i*3:(i*1)*3] for i in range(3)]
- **assigning reductor: multiple assignment allows us to assign multiple variables (same or different) at the same time in one line of code -> Spongebob=Patrick=Sandy=Squidward=30**
- **augmented assignment operator: writing same code in a shorter form -> x += 3**

- **string methods:** len(str),str.find("B"),str.capitalize(),str.lower()
- **slicing a salami:** meat="salami" **->** print("meat[2:5]",meat[2:5]) **||** lam **//** meat[:3] **||** sal **//** meta[2:] **||** lami **//** meat[-3:] **||** ami **=> slice a list same as a string**

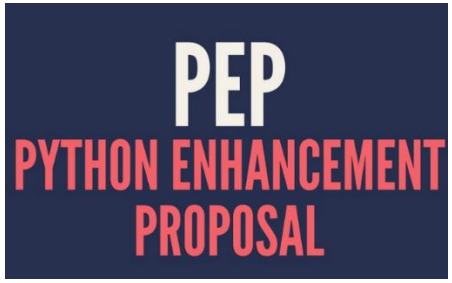
- **math functions:** import math + round(pi),math.ceil(pi),math.floor(pi),math.sqr(420),max(x,y,z),min(x,y,z)
  -> pow(pi,2) **-> raise base number to a power so you need a base and exponent**
  -> abs(pi) **-> abs will tell you how far a number as far from 0**
- **arithmetic operators:**
  **** = to the power 10 ** 3 = 1000**
  // = will give just the integer division

- **time delay: import time** **->** container="lulu" **-> time.sleep(1)** **->** print(container) **# so 1 will 1 second**

- **python:** programming without **boilerplate (prepared code)**
- **run our script:** run our program
- **dry:** don't repeat the same code **=>** that will lead to errors
- **one liner code:** one line code that makes code easy
- **synthetic sugar:** code simplified for user behind the scenes through objects
- **indentation:** std 4 empty space at beginning of paragraph used for blocks instead of curly braces {}
- **":":** means it will be a suit of that block, will tell you that next line will be special, that indentation is needed, python will break without ":"
- **module:** each file is called module, different files responsible for logic or drawing on the screen = keep organized = just python files with **.py extension**
  -> a separate file with a reusable code (like a supermarket, it has different sections like fruits, cleaning products, junk food -> each section in the supermarket is like a module in python)
  -> to look for the module in google you will have to type **->** python 3 math module
- **import:** make all those functions accessible in my script so we can call them
  -> import: when we call import **....** it goes to the **....** package in python, and it says all these functions inside the package, make these accessible in our script so we can call these functions
- **variable:** a container for a value, behaves as the value that it contains
- **initializing a var:** specifying an initial value to assign to it (before it is used), var that does not have a defined value it is not initialized so it **CANNOT** used until it is assigned a value
- **string:** is actually a list of characters
- **concatenating:** joining 2 strings
- **to refactor (refactor the code):** to rename
- **5 + 6 => 5 & 6** are operands and **+** is operator
- **x = x+1 =>** the **=** is not equality it is assignment
- **print('*' * 10):** what is inside the () is called an **expression** so it is a piece of code that produces a value,
  -> so when the **python interpreter tries to execute the line first it evaluates the code in () so -> our expression will produce 10 * and then * will be printed on the terminal**
- **==** is sign of comparing
- **=** is just an assign operator
- **0 index :** is like the ground level in a hotel, basement is -1 and first level is 1
- **element number 0:** index [0]
- **print(board[0]):** print board at 0
- **a = int(a) ->** a gets int **->** so take whatever a was, converted it to an integer and put it back in a
- **price = 10:** so 10 will be the box and price (identifier) will be the label on the box
  -> so in other words **price = 10 is identifier = value and before 10 gets stored**, first it gets converted to binary representation so 0 and 1 **->** then we can access anywhere in the program what is in the box
- **if statement:** a block of code that will execute if its condition is true, it matters the order you put the commands
- **if user_letter in alphabet - used letters:** if user letter in alphabet but haven't used yet
- **logical operators, compound conditions (and,or,not):** used to check if two or more conditional statements are true
- **sentry variable in for loop:** is the key to open the door in a list
- **while guess!=random_number:** while (expression) **->** Then it will iterate over some things
- **while tripFinished !="YES":** **->** so anything but yes then loop
- **list:** list of items in single variable **BUT** list takes lot of memory
- **built in functions:** print and input are part of built in functions for common tasks, those functions, they are like the buttons on your remote control and **with () you are calling (executing) that function like pressing the button on the remote control**
- **when define the function:** give it a name
- **functions arguments:** variable passed from the caller to the function
- **magic methods:** __add__(),__sub__(),__mul__()
- **class Player: ->** def __init__(self,letter) = create base player class and in this class and we're going to initialize it with the letter that the player is going to represent
- **object oriented:** python is completely object oriented and not **"statistically typed"**
- **objects:** encapsulation of variables and functions into a single entity **->** objects get their var and func from classes **->** classes are essentially a template to create your objects
- **inheritance:** enables a new class to take the properties of an existing class -> the existence of these base classes and derived classes makes managing relationships between classes a lot easier
  -> inheritance: we initialize the super class, so it is going to call the initialization in the super class
- **polymorphism:** enables you to redefine a function in a derived class ex. animal class has a function called make sounds and derived class called dog and cat should make different sounds so we will redefine same function differently
  -> polymorphism: Duck Typing, Operator overloading, method overloading and overriding
- **abstraction:** using function without worrying how they work, hiding unnecessary parts of code to look at the big picture
- **encapsulation:** put it in a bag and give it a name (function or list for data together and give it a name)

-> an object has its private and public variables private; variables aren't accessible from outside of the object's class

- **keyword argument:** combination of the paramter name followed by its value in the caller of the function, so order of targeted parameters won't be an issue and it increases readability of the code
  -> keyword arguments must always come AFTER positional arguments -> greet_user = (last_name = "Smith", first_name = "John")
- **positional arguments vs keyword arguments:** in positional arguments you might understand it is about first and last name but if you have numbers, you won't understand what those numbers refer to
  -> so, we use keyword arguments (prefix the arguments with the name of their parameter)



What is a PEP?

- PEP is like amendments to the constitution for a country which similarly determines how a country is governed each PEP build on the next one or the previous one and makes changes sort of like the law of python
  - So, we start with PEP1 and we grow from there, example peppy (PEP 8 Style Guide for Python Code – author is Guido Von Rossum). PEP 20 The Zen of Python (it's what you get when you type import this in the your python repple – author is Tim Peters, PEP 566 Metadata for Python Software Packagees – author is Dustin Ingram)
- To submit a PEP:
  1. **Draft**
  2. **Acceptance**
  3. **Implementation**
- PEP stands for Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment
  - The PEP should provide a concise technical specification of the feature and a rationale for the feature
- We intend PEPs to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python
  - The PEP author is responsible for building consensus within the community and documenting dissenting opinions
- Because the PEPs are maintained as text files in a versioned repository, their revision history is the historical record of the feature proposal
  - This historical record is available by the normal git commands for retrieving older revisions, and can also be browsed on GitHub

PEP Audience

- The typical primary audience for PEPs are the core developers of the CPython reference interpreter and their elected Steering Council, as well as developers of other implementations of the Python language specification
- However, other parts of the Python community may also choose to use the process (particularly for Informational PEPs) to document expected API conventions and to manage complex design coordination problems that require collaboration across multiple projects

PEP Types

- There are three kinds of PEP:
  1. A **Standards Track** PEP describes a new feature or implementation for Python. It may also describe an interoperability standard that will be supported outside the standard library for current Python versions before a subsequent PEP adds standard library support in a future version
  2. An **Informational** PEP describes a Python design issue, or provides general guidelines or information to the Python community, but does not propose a new feature. Informational PEPs do not necessarily represent a Python community consensus or recommendation, so users and implementers are free to ignore Informational PEPs or follow their advice
  3. A **Process** PEP describes a process surrounding Python, or proposes a change to (or an event in) a process. Process PEPs are like Standards Track PEPs but apply to areas other than the Python language itself. They may propose an implementation, but not to Python's codebase; they often require community consensus; unlike Informational PEPs, they are more than recommendations, and users are typically not free to ignore them. Examples include procedures, guidelines, changes to the decision-making process, and changes to the tools or environment used in Python development. Any meta-PEP is also considered a Process PEP

- **spacing:** always add 2 lines after the def as per PEP8

1. Python tutorial for beginners

****

- install from python.org/download
- download PyCharm > community > check launcher and add launcher to the path
-file > python file > name it

```
print('hello world')
print("it's really good")
```

//

2. Variables

****

 # variable is a container for a value. Behaves as the value that it contains

```
name = "Superman"
print("Hello "+name)
print(type(name))//checking data type

#first_name="Bro"//use # to deactivate
#last_name="Code"
#full_name=first_name+ " " +last_name
#print("full_name")

age=21
age=age+1
age+=1
print(type(age))

print("your age is: :+age) // error
print("your age is: :+str(age)) // ok
# So you have to convert to be same datatype or it will not work

height=250.5
print(height)
print(type(height))
print("your height:"+str(height)+"cm")
```

```
human=True
print(human)
print(type(human))
print("Are you human : "+str(human))
```

//

3. Multiple Assignment

****

 # Multiple assignment = allows us to assign multiple variables at the same time in one line of code

#Ex1:

```
#name="Bro"
#age=21
#attractive=True
name,age,attractive="Bro",21,True // same as above but grouped together, you have to respect the order
print(name)
print(age)
print(attractive)
```

#Ex2:

```
#Spongebob=30
#Patrick=30
#Sandy=30
#Squidward=30
Spongebob=Patrick=Sandy=Squidward=30
print(Spongebob)
print(Patrick)
print(Sandy)
print(Squidward)
```

//

4. String Methods

****

```
name = "Bro"
#print(len(name))
print(name.find("B"))
print(name.capitalize())
print(name.upper())
print(name.lower())
print(name.isdigit())
print(name.isalpha)
print(name.count("o"))
print(name.replace("o","a")
print(name*3)
```

//

5. Type Cast

****

 # Type Casting = convert the data type of a value to another data type

```
x = 1 #int
y = 2.0 #float
z = "3" #str

y=int(y) #permanent
z=int(z*3)
x=float(x)
print(x)
```

```
print(int(y)) #temporary change
print(z)
print("x is"+str(x)) # so when you print you have to convert t have same values
```

//

6. User Input

****

```
name = input("what is your name?: ")
age = int(input("what is your age?"))
height=float(input("how tall are you?"))
age = age +1
print("hello "+name)
print("your age is:"+str(age))
print("you are"stra(height)+"cm tall")
```

//

7. Math Functions

****

```
import math // math module
pi=-3.14
x=1
y=2
z=3

print(round(pi))
print(math.ceil(pi))
print(math.floor(pi))
print(abs(pi))//abs will tell you how far a number as far from 0
print(pow(pi,2))//raise base number to a power so you need a base and exponent
print(math.sqr(420))
print(max(x,y,z))
print(min(x,y,z))
```

//

8. String Slicing

****

```
# slicing = create a substring by extracting elements from another string
# indexing[] or slice()
# [start:stop:step]

name = "Bro Code"
```

indexing:

```
first_name=name [:3]
last_name=name[4:]
funky_name=name[0:8:2] / /result BoCd = so it will print every 2 characters you can write [::3]
reversed_name=name[::-1] // name to be printed in reverse
print(first_name) // wil print just B
first_name=name[0:2] // BR because the stoping index is not inclusive so you have to write [0:3] or [:3]
```

slice:

```
website="http//google.com"
slice = slice(7,-4)
print(website[slice])
```

//

9. If Statements

****

 # if statement = a block of code that will execute if it's condition is true

```
age=int(input("How old are you?: "))
if age >=18:
print("you are an adult!")
elif age ==100:
print("you are a century old") // will print you are an adult because the first one got executed, so you have to move this up, so it matters the order you put the commands
elif age <0:
print("you haven't been born yet")
else:
print("you are a child")
```

//

10. Logical Operators

****

 # logical operators (and,or,not) = used to check if two or more conditional statements are true

```
temp=int(input("what is the temperature outside?: "))

if temp >=0 and temp <=30:
print("the temperature is good today"!")
print("go outside!")

elif temp <0 or temp >30:
print("the temperature is bad today!")
print("stay inside")
```

for not:

```
you put all in () and not in font
if not(temp >=0 and temp <=30):
print("the temperature is bad today!")
print("stay inside")

elif not(temp <0 or temp >30):
print("the temperature is good today"!")
print("go outside!")
```

//

11. While Loops

****

 # while loops = a statement that will execute its block of code, as long as it's condition remains true

```
while 1==1:
print("Help! I'm stuck in a loop!")

name = ""
while len(name) == 0:
name = input("Enter  your name")
print("Hello "+name)
```

Same you can write:

```
name = None
while not name:
name=input("Enter your name: ")
print("Hello "+name)
```

//

12. For Loops

****

 # for loop = a statement that will execute its block of code a limited amount of times
 # while loop = unlimited for loop // for loop = limited

```
for i in range(10):
print(i+1)

for i in range (50,100+1): // print from 50 to 100, we added 1 as 100 is not inclusive so adding 1 will give counting from 50 to 100
print(i)

for i in range (50,100+1,2): // 2 is the step, so from 2 to 2 will count
print(i)

for i in "Bro Code": // will print bro code with letters below each other
print(i)
```

Ex countdown:

```
import time
for seconds in range (10,0,-1): // -1 to make it countdown
print(seconds)
time.sleep(1) / /will sleep for 1 second
print("Happy New Year!")
```

//

13. Nested Loops

****

 # nested loops = The "inner loops" will finish all of it's iterations before finishing one iteration of the "outer loop" so loop in a loop if it's while or for!

```
rows = int(input("How many rows?: ")) // for outer loop
columns = int(input("How many columns?: ")) // for inner loop
symbol=input("Enter a symbol to use: ")

for i in range(rows):
  for j in range(columns):
     print(symbol, end="")
  print()
```

//

14. Break Continue Pass

****

 # Loop Control Statements = change a loops execution from its normal sequence

 # break = used to terminate the loop entirely
 # continue = skips to the next iteration of the loop
 # pass = does nothing, acts as a placeholder

```
while True:
  name=input("enter your name: ")
  if name !="":
    break

phone_number="123-456-7890"
for i in phone_number:
if i=="-":
continue
print(i, end="")

for i in range(1,21):
 if i ==13;
   pass // it does nothing it passes as placeholder
else:
```

```
print(i)
```

//

15. Lists

****

 # list = used to store multiple items in a single variable

```
food=["pizza","hamburger","hotdog","spaghetti']
print(food)
food[0]="sushi
print(food[0])

for x in food:
  print(x)

food.append("ice cream")
food.remove("hotdog")
food.pop() // so pop will remove last element
food.insert(0,"cake")
food.sort() // will sort in alphabetical
food.clear() // will remove all elements of a list
```

//

16. 2D Lists

****

 # 2D lists = a list of lists

```
drinks=["coffee","soda","tea"]
dinner=["pizza", "hamburger","hotdog"]
dessert=["cake","ice cream"]

food=[drinks,dinner,dessert]

print(food)

print(food[0][0])
```

//

17. Tuples

******

#tuple = collection which is ordered and unchangeable
used to group together related data

```
student = ("Bro", 21,"male")
print(student.count("Bro"))
print(student.index("male"))

for x in student:
print(x)

if "Bro" in student:
print("Bro is here")
```

//

18. Sets

****

 # set = collection which is unordered, unindexed.No duplicate values.

```
utensils={"fork","spoon","knife"}
dishes={"bowl","plate","cup","knife"}

utensils.add("napkin")
utensils.remove("fork")
utensils.clear()
utensils.update(dishes)
dishes.update(utensils)

dinner_table=utensils.union(dishes)

for x in utensils:
print(x)

print(utensils.difference(dishes)) // what utensils have that dishes don't
print(utensils.intersection(dishes)) // what they have in common
```

//

19. Dictionaries

****

```
# dictionary = A changeable, unordered collection of unique key: value pairs
# Fast because they use hashing, allow us to access a value quickly

capitals = {'USA':'Washington DC',
            'India':'New Delhi',
            'China':'Beijing',
            'Russia':'Moscow'}
capitals.update({'Germany':'Berlin'})
capitals.update({'USA':'Las Vegas'})
capitals.pop('China')//delete China
capitals.clear()//delete all dictionary

print(capitals['Russia'])
print(capitals['Germany']) // error
print(capitals.get('Germany')) // will say None without error
print(capitals.keys())
print(capitals.values())
print(capitals.items())

for key,value in capitals.items():
print(key,value)
```

//

20. Indexing

****

```
# index operator [] = gives access to a sequence's element (str,list,tuples)

name = "bro Code"

if (name[0].islower()):
name = name.capitalize()

first_name=name[0:3].upper() //you can specify start and ending position => you can use [:3]
last_name=name[4:].lower()
print(name)
print(first_name)
print(last_name)
last_character=name[-1] //-1 is the last character
print(last_character)
```

//

21. Functions

****

 # function = a block of code which is executed only when it is called

```
def hello(name): // name is a parameter that will match bro that is an argument
print("hello!" +name)
print("Have a nice day")
hello("Bro") // bro is an argument and you need a matching parameters//information sent to a function
name="bro" // so you can do it like this also

def hello(first_name,last_name,age)
print("hello"+first_name+" "+last_name)
print("you are"+str(age)
hello("Bro","Code",21)
```

CZ:so we can pass information as argument but we need to pass a matching number of parameters to receive this argument

//

22. Return Statement

****

 # return statement = Functions send Python values/objects back to the caller.
 # These values/objects are known as the function's return value

```
def multiply(number1,number2):
result=number1*number2
return result

x=multiply(6,8)
print(multiply(6,8)) or print (x) if i have allocated to x
```

Or easier:

```
def multiply(number1,number2):
return number1*number2
```

//

23. Keyword Arguments

****

 # keyword arguments = arguments preceded by an identifier when we pass them to a function
 # the order of the arguments doesn't matter, unlike positional arguments
 # Python knows names of the arguments that our function receives

```
def hello(first,middle,last):
print("Hello "+first+" "+middle+" "+last)

hello("Bro","Dude","Code") // will print hello Bro Dude Code
hello("Code","Dude","Bro") // will print hello Code Dude Bro
hello(last="Code",middle="Dude",last="Bro") // by passing identifier for the arguments it will print hello Bro Dude Code
```

//

24. Nested Function Calls

****

Nested Functions Calls = function calls inside another function calls
 # innermost function calls are resolved first
 # returned value is used as argument for the next outer function

```
num=input("Enter a whole positive number: ")
num=float(num)
num=abs(num)
```

```
num=round(num)
print(num)
```

you can write the above as through nested function calls:

```
print(round(abs(float(input("Enter a whole positive number: ")))))
```

//

25. Variable Scope

****

 # scope = The region that a variable is recognized
 # A variable is only available from inside the region it is created.
 # A global and locally scoped versions of a variable can be created

```
name = "Bro" # global scope(available inside & outside functions)
def display_name():
name = "Code" // this variable has a local scope because it is declared inside a function = local variable = local scope = so you cannot access it from outside
print(name)

print(name) //error => once added as global it won't give error
```

Obs.You can have variable to be global and local with the same name
If you change "code" not to be there => LEGB (local enclosing global built-in)

//

26. *Args parameters

****

***args = parameter that will pack all arguments into a tuple**
 # useful so that a function can accept a varying amount of arguments

```
def add(num1,num2): // args name is not important you can put anything but * is important as it means packing
sum = num1+num2
return sum
print(add(1,2))
```

For having any arguments added we use * and name of the pack ex. *args

```
def add(*args): // args name is not important you can put anything but * is important as it means packing
sum = 0
args[0]=0 // will give error if you try to change
args=list(args)
args[0]=0 // will not give error as we cast it
for i in args:
sum+=i
return sum
print(add(1,2,3))
```

//

27. **Kwargs

****

 # **kwargs = parameter that will pack all arguments into a dictionary
 # useful so that a function can accept a varying amount of keyword arguments

```
def hello(first, last):
print("Hello" + first +"  " + last)
hello(first="Bro",last="Code")

def hello(**kwargs): // so name of our dictionary is kwargs
print("Hello" + kwargs['first'] +"  " + kwargs['last']) // this will print Hello Bro Code
print("Hello",end=" ")
```

```
for key,value in kwargs.items():
print(value,end=" ")
hello(first="Bro",,middle="Dude",last="Code") // will print Hello Bro Dude Code
```

obs: kwargs = means keyword arguments but you can use any word

//

28. String Format

****

 # str.format() = optional method that gives users more control when displaying output

```
animal = "cow"
item="moon"
print("The  "+anima+" jumped over the "+item)

it can be written:
print("The {} jumped over the {}".format("cow","moon"))
print("The {} jumped over the {}".format(animal,item"))
print("The {0} jumped over the {1}".format(animal,item")) # positional argument
print("The {animal} jumped over the {item}".format(animal="cow",item="moon")) # keyword argument

text="The {} jumped over the {}"
print(text.format(animal,item))

name="Bro"
print("Hello, my name is {}".format(name))
print("Hello, my name is {:10} nice to meet you".format(name)) # it will add taping : Hello, my name is Bro          nice to meet you

you can use {:10} or :<10 or :> or :^

number = 3.14159
print("The number pi is {:.2f}".format(number))

number=1000
print("The number pi is {:,}".format(number))
print("The number pi is {:b}".format(number)) # display number in binary
print("The number pi is {:o}".format(number)) # display number in octal
print("The number pi is {:x}".format(number)) # display number in octal
print("The number pi is {:E}".format(number)) # display number in scientific
```

//

29. Random Numbers

****

 # pseudo-random

```
import random # import random
x=random.randint(1,6)
y=random.random()
print(x)
print(y)

myList=['rock','paper','scissors']
z=random.choice(myList)
print(z)

cards=[1,2,3,4,5,6,7,8,9,"J","Q","K","A"]
random.shuffle(cards)
print(cards)
```

//

30. Exception Handling

****

 # exception = events detected during execution that interrupt the flow of a program

```
try:
  numerator = int(input("Enter a number to divide: "))
  denominator=int(input("Enter a number to divide by: "))
  result=numerator/denominator
except Exception:
  print("something went wrong :(")
```

but it is a good practise to try/catch for specific situation:

```
try:
  numerator = int(input("Enter a number to divide: "))
  denominator=int(input("Enter a number to divide by: "))
  result=numerator/denominator
except ZeroDivisionError as e: # as e is not mandatory you can remove it,
  print(e) # so it will print the error first, so you use as e to print the error
  print("You can't divide by 0")
except ValueError as e:
  print("Enter only numbers plz")
#except Exception as e:
  #print("something went wrong :(")

else:
  print(result)
finally:
  print("This will always execute)
```

//

31. File Detection

****

```
import os
path="c:\\Users\\khali.alqutob\\desktop\\test.txt"
if os.path.exists(path):
print("that location exists!")
 if os.path.isfile(path):
print("That is  a file")
else:
print("That location does not exists!")

path="c:\\Users\\khali.alqutob\\desktop\\folder"
if os.path.exists(path):
print("that location exists!")
 if os.path.isfile(path):
print("That is a file")
elif os.path.isdir(path):
print("That is a directory")

else:
print("That location does not exists!")
```

//

32. Read a File

****

```
with open('test.txt') as file: # if it is not in the folder then you will have to put full path//it will read and close the file
print (file.read())
```

try: **// use try and catch to print errors**

```
with open('test.tx') as file:
print(file.read())
except FileNotFoundError:
print("That file was not found:")
```

//

33. Write a File

****

```
text="Yooooooo\n this is some text\nHava good one" // use\n for enter
with open('test.txt','w') as file: // use a to append text instead of w (so r,w,a are modes)
file.write(text)
```

//

34. Copy a File

****

```
# copyfile() = copies contents of a file
# copy() = copyfile() + permission mode+destination can be a directory
# copy2() = copy() + copies metadata (file's creation and modification times)

import shutil
shutil.copyfile('test.txt','C:\\Users\\BroCode\\Desktop\\copy.txt') # 2 arguments (source and destination) - you put where the file is, if same folder ok if out then full path
shutil.copy2('test.txt','C:\\Users\\BroCode\\Desktop\\copy.txt')
```

//

35. Move a File

****

```
import os

source = "test.txt"
destination = " C:\\Users\\Cakeow\\Desktop\\test.txt"
try:
if os.path.exists(destination)
print("There is already a file there")
else:
os.replace(source,destination)
print(source+"was moved")
except FileNotFoundError:
print(source" was not found")

source = "folder"
destination = " C:\\Users\\Cakeow\\Desktop\\folder"
try:
if os.path.exists(destination)
print("There is already a file there")
else:
os.replace(source,destination)
print(source+"was moved")
except FileNotFoundError:
print(source" was not found")
```

//

36. Delete a File

****

```
import os

os.remove('test.txt') # if it is somewhere else(not in the file) then you will have to list the full path or pass a variable

path="test.txt" # you can assign a variable
```

os.remove(path)

But! if you will try to delete a file that does not exist then you will have an exception so optional you can do an exception handling

```
path="test.txt"
try:
os.remove(path)
except FileNotFoundError:
print("That file was not found")

path="empty_folder"
try:
# os.remove(path) # you cannot delete folder with this function as you cannot delete an empty folder
os.rmdir(path) # short for remove directory and this will work
except FileNotFoundError:
print("That file was not found")
except PermissionError:
print("You do not have permission to delete that")
else:
print(path+" was deleted")

import os
import shutil
path="folder" // if you have a file in the folder
try:
# os.remove(path) # you cannot delete folder with this function as you cannot delete an empty folder
# os.rmdir(path) # short for remove directory and this will work but will not work if there are files inside
shutil.rmtree(path) # short for remove tree, so this will remove the folder and the files inside but you will have to import the shutil module => in total 3 functions to delete
except FileNotFoundError:
print("That file was not found")
except PermissionError:
print("You do not have permission to delete that")
except OSError:
print("You cannot delete that using that function")
else:
print(path+" was deleted")
```

//

37. Modules

****

➢ As you go forward you must organize your code into functions, classes and modules

```
""" import a module """
import converters
print(converters.kg_to_lbs(70))

""" but to import a specific function """
from converters import kg_to_lbs
kg_to_lbs(100) # -> so no need to use print and .converters
```

➢ python comes with a standard library that contains several modules for common tasks such as sending emails, working with date and time, generating random values and passwords, modules already built-in modules so you will not do it from scratch -> go to google -> python 3 module index (because python 2 and 3 are different modules)

➢ importing from another module with * will run everything
from critter import * # -> so will run def main() but also run main() -> importing with * will run everything

➢ to find out if you are in the main or in different module importing the main use the following:

```
print(__name__)
```

➢ or as condition to tell you if it is the main or not:

```
if__name__=='__main__':
  print("running this module directly")
```

```
else:
  print("running other module indirectly")
```

➢ **or you can be sneaky and run a command which you want like print using this (in a way will act as a Boolean):**

```
if __name__=='__main__':
  print("hello")
```

//

38. Rock, paper, scissors game

****

```
import random

while True:
  choices = ["rock","paper","scissors"]

  computer = random.choice(choices)
  player = None

  while player not in choices:
    player = input("rock, paper, or scissors?: ").lower()

  if player == computer:
    print("computer: ",computer)
    print("player: ",player)
    print("Tie!")

  elif player == "rock":
    if computer == "paper":
      print("computer: ", computer)
      print("player: ", player)
      print("You lose!")
    if computer == "scissors":
      print("computer: ", computer)
      print("player: ", player)
      print("You win!")

  elif player == "scissors":
    if computer == "rock":
      print("computer: ", computer)
      print("player: ", player)
      print("You lose!")
    if computer == "paper":
      print("computer: ", computer)
      print("player: ", player)
      print("You win!")

  elif player == "paper":
    if computer == "scissors":
      print("computer: ", computer)
      print("player: ", player)
      print("You lose!")
    if computer == "rock":
      print("computer: ", computer)
      print("player: ", player)
      print("You win!")

  play_again = input("Play again? (yes/no): ").lower()

  if play_again != "yes":
    break
```

```
print("Bye!")
```

//

39. Quiz game

****

python quiz game project tutorial example explained

#python #quiz #game

```
# --------------------------
def new_game():

    guesses = []
    correct_guesses = 0
    question_num = 1

    for key in questions:
        print("--------------------------")
        print(key)
        for i in options[question_num-1]:
            print(i)
        guess = input("Enter (A, B, C, or D): ")
        guess = guess.upper()
        guesses.append(guess)

        correct_guesses += check_answer(questions.get(key), guess)
        question_num += 1

    display_score(correct_guesses, guesses)

# --------------------------
def check_answer(answer, guess):

    if answer == guess:
        print("CORRECT!")
        return 1
    else:
        print("WRONG!")
        return 0

# --------------------------
def display_score(correct_guesses, guesses):
    print("--------------------------")
    print("RESULTS")
    print("--------------------------")

    print("Answers: ", end="")
    for i in questions:
        print(questions.get(i), end=" ")
    print()

    print("Guesses: ", end="")
    for i in guesses:
        print(i, end=" ")
    print()

    score = int((correct_guesses/len(questions))*100)
    print("Your score is: "+str(score)+"%")

# --------------------------
def play_again():
```

```
    response = input("Do you want to play again? (yes or no): ")
    response = response.upper()

    if response == "YES":
        return True
    else:
        return False
# -------------------------


questions = {
 "Who created Python?: ": "A",
 "What year was Python created?: ": "B",
 "Python is tributed to which comedy group?: ": "C",
 "Is the Earth round?: ": "A"
}

options = [["A. Guido van Rossum", "B. Elon Musk", "C. Bill Gates", "D. Mark Zuckerburg"],
        ["A. 1989", "B. 1991", "C. 2000", "D. 2016"],
        ["A. Lonely Island", "B. Smosh", "C. Monty Python", "D. SNL"],
        ["A. True","B. False", "C. sometimes", "D. What's Earth?"]]

new_game()

while play_again():
    new_game()

print("Byeeeeee!")

# -------------------------
```

//

40. Object Oriented Programming OOP

****

 # mimic real world through attributes = is/has ex.name, age, height & methods = action ex.eat,sleep, make youtube videos
 # we need to create a class that will be the blueprint to describe the distinct attributes and methods that the object will have
 # class name will be first letter capital
 # to import class if it is in another module you write: from car import Car

```
class Car: # it is called the constructor
def__init__(self,make,model,year,color): # this is the constructor that we need it or it will not work
self.make=make      #instance variable
self.model=model    #instance variable
self.year=yea       #instance variable
self.color=color      #instance variable

def drive(self): # we need 1 argument, self refers to the object that uses this method
print("This car is driving")
def stop(self):
print("This car is stopped")

car_1=Car("Chevy","Corvette",2021,"Blue")
car_2= Car("For","Mustang",2022,"red")

print(car_1.make)
print(car_1.model)
print(car_1.year)
print(car_1.color)

car_1.drive()
```

car_1.stop()

car_2.drive()
car_2.stop()

you can write
def drive(self):
print("This "+self.model+" car is driving"

//

41. Class Variables
****
from car import Car

car_1=Car("Chevy","Corvette",2021,"Blue")
car_2= Car("For","Mustang",2022,"red")

class Car:

wheels = 4 **# class variable - created within the class but outside the constructor - you can set a default value for all instances of this class for all unique values created and can be changed later if needed**

def__init__(self,make,model,year,color):
self.make=make #instance variable - declared inside a constructor and given unique values
self.model=model #instance variable - declared inside a constructor and given unique values
self.year=yea #instance variable - declared inside a constructor and given unique values
self.color=color #instance variable - declared inside a constructor and given unique values

car_1.wheels = 2

print(car_1.wheels) # will print 2
print(car_2.wheels) # will print 4

Car.wheels = 2
Then it will print all instances as 2

or you can do it like this:
print(Car.wheels)

//

42. Inheritance
****
 # classes can inherit usually attributes and methods from other classes like parent and a child like inheriting genes from the parent and classes they can have children (family)

class Animal:
alive = True
def eat(self):
print("This animal is eating")
def sleep(self):
print("This animal is sleeping")

class Rabbit(Animal): **# so rabbit is the child class and animal is the parent class, # use pass if you want to add later info for the class**
def run(self):
print("This rabbit is running")
class Fish(Animal):
def swim(self):
print("This fish is swimming")
class Hawk(Animal):
def fly(self):
print("This hawk is flying")

```
rabbit=Rabit() # create object from this classes, # each of those children classes will inherit from parent class
fish=Fish()
hawk=Hawk()

print(rabbit.alive)
fish.eat()
hawk.sleep()

rabbit.run()
fish.swim()
hawk.fly()
```

//

43. Multilevel Inheritance

****

```
# multi-level inheritance = when a derived (child) class inherits another derived (child) class
# Grandparent > Parent > Child = family tree that inherit like DNA

class Organism:
alive = True # attribute of alive
class Animal(Organism):
def eat(self):
print("This animal is eating")
class Dog(Animal):
def bark(self):
print("This dog is barking")

dog=Dog() # create an object
print(dog.alive)
dog.eat()
dog.bark()
```

//

44. Multiple Inheritance

****

```
# multiple inheritance = when a child class is derived from more than one parent class = bigger fish eating big fish eating smaller fish

class Prey"
def flee(self):
print("This animal flees")

class Predator:
def hunt(self):
print("This animal is hunting")

class Rabbit(Prey):
pass
class Hawk(Predator):
pass
class Fish(Prey, Predator)

rabbit=Rabbit()
hawk=Hawk()
fish=Fish()

rabbit.flee()
hawk.hunt()
fish.flee()
fish.hunt()
```

//

45. Method Overriding

****

```
class Animal:
def eat(self):
print("This animal is eating")
class
Rabbit(Animal):
def eat(self):
print("This rabbit is eating a carrot!") # So this will override the main method from parent class

rabbit=Rabbit()
rabbit.eat()
```

=> 2 methods with same name and ar/par used by different classes

```
class A:
def show(self):
print("in A show")

class B(A):
print("in B Show")

a1=A()
a1.show()
a1=B()
a1.show()
```

//

46. Method Chaining

****

 # method chaining = calling multiple methods sequentially
 # Each call performs an action on the same object and returns self

```
class Car:
def turn_on(self):
print("You start the engine")
return  self # you have to add this for Method Chaining
def drive(self):
print("You drive the car")
return self
def brake(self):
print("You step on the brakes")
return self
def turn_off(self):
print("You turn off the engine")
return self

car=Car()
car.turn_on().drive().
car.brake().turn_off()

car.turn_on\
.drive()\ # \ is line continuation
.brake()\
.turn_off()
```

//

47. Super Function

****
 # super() = Function used to give access to the methods of a parent class.
 # Return object of a parent class when used

```
class Rectangle:
def __init__(self, length, width): # so Square and Cube had same as below => we write in the parent then we access it through super
self.length=length
self.width=width

class Square(Rectangle):
def __init__(self, length, width):
super().__init__(length,width)

def area(self):
return self.length*self.width

class Cube(Rectangle):
def __init__(self, length,width, height):
super().__init__(length,width)
self.height=height

def volum(self):
return self.length*self.width*self.height

square=Square(3, 3)
cube = Cube(3,3,3)

print(square.area())
print(cube.volume())
```

//

48. Abstract Classes

****
 # Prevents a user from creating an object of that class // you cannot create another vehicle! just to use what is inside already created!
 # + compels a user to override abstract methods in a child class // it will force you to put same method in both car and motorcycle or it will give error

 # abstract class = a class which contains one or more abstract methods.
 # abstract method = a method that has a declaration but does not have an implementation
 # abstract classes are like ghost classes like a template, an idea

```
from abc import ABC, abstractmethod //you have to add this when creating abstract class, abstract base class
class Vehicle(ABC): // so adding() and inside ABC it will inherit abc
@abstractmethod // add this also so it will work
def go(self):
pass
@abstractmethod
def stop(self):
pass
class Car(Vehicle):
def go(self):
print("You drive the car")
def stop(self)
print("The car stopped")
class Motorcycle(Vehicle):
def go(self):
print("You ride the motorcycle")
def stop(self)
print("The motorcycle stopped")

#vehicle = Vehicle()
car = Car()
```

```
motorcycle = Motorcycle()

#vehicle.go()
car.go()
motorcycle.go()

car.stop()
motorcycle.stop()
```

//

49. Objects as Arguments

****

```
class Car:
color = None

class Motorcycle:
color=None

def change_color(car, color ): // car in both can be anything but both words have to match
car.color=color

car_1=Car()
car_2=Car()
car_3=Car()

bike_1=Motorcycle()

change_color(car_1,"red")
change_color(car_2,"white")
change_color(car_3,"blue")

change_color(bike_1,"black")

print(car_1.color)
print(car_2.color)
print(car_3.color)

print(bike_1.color)
```

Cz. So we can pass objects as arguments to a function much like variables but the type of objects that we will pass in will be limited to the required attributes and methods that the object or class might have

//

50. Duck Typing

****

```
# Duck typing = concept where the class of an object is less important than the methods/attributes
# class type is not checked if minimum methods/attributes are present
# "If it walks like a duck, and it quacks like a duck, then it must be a duck."

class Duck:
def walk(self):
print("This duck is walking")
def talk(self):
print("This duck is qwuacking")

class chicken(self):
def walk (self):
print("This chicken is walking")
def talk(self):
print("This chicken is qwuacking")
```

```
class Person():
def catch(self, duck):
duck.walk()
duck.talk()
print("You caught the critter!")
```

//creating objects

```
duck - Duck()
chicken=Chicken()
person=Person()
```

person.catch(duck) // if you change the parameter to chicken it will still give same result but if you remove the walk method from chicken then it will give error
CZ:so even if you pass a different parameter if it talks and walks then it is the same, if talk or walk is missing then it will give error so chicken will not be the same as duck

//

51. walrus operator:=

****

 # walrus operator :=
 # New to Python3.8
 # assignment expression aka walrus operator
 # assigns values to variables as part of a larger expression

ex.1

```
happy = True
print(happy)
```

//or we can print it like this:

```
print(happy:=True)
```

ex.2

```
foods=list() // list() creates a list object, it is ordered and changable
while True:
food=input("What food do you like?: ")
if food=="quit":
break
foods.append(food)
```

//or we can print it like this:

```
foods=list()
while food:=input("What food do you like?:") !="quite":
foods.append(food)

print(foods)
```

//

52. Functions to Variables

****

def hello():
print("Hello")

ex1:

hello(hello) // this will print the memory address of this function, where is located in computer's memory in hexadecimal = like street address, this changes
hi=hello
hello()
hi() // so hello or high is the same, so variable hi took the stuff from the function hello, like an alias, like the function hello has 2 names

ex2:

```
say=print
say("Whoa! I can't believe this works!") // so print and say will do the same thing!
```

//

53. Higher Order Functions

****

 # Higher Order Function = a function that either:
 # 1. accept a function as an argument
 # or
 # 2. returns a function (In Python, functions are also treated as objects)

ex for 1

```
def loud(text): //this will accept string as an argument
return text.upper()

def quiet(text):
return text.lower()

def hello(func):
text=func("Hello")
print(text)

hello(loud)
hello(quiet)
```

ex for 2 // 10/2=5 => dividend/divisor=quotient

```
def divisor(x):
 def dividend(y):
  return y/x
 return dividend

divide = divisor(2)
print(divide(10))
```

//

54. Lambda

****

 # lamdba function = function written in 1 line using lambda keyword
 # accepts any number of arguments, but only has one expression.
 # (think of it as a shortcut)
 # (userful if needed for a short period of time, throw-away)
 # lambda parameters: expression // this is the syntax

ex:

```
def double(x):
return x*2
print(double(5))
```

or shorter using lambda:// so you type lambda, your parameters: then your expression

```
double = lambda x:x *2
multiply = lambda x,y: x*y // we passed 2 arguments
print(double(5))
print(multiply(5,6))
add=lambda x,y,z:x+y+z
print(add5,6,7))
full_Name=lambda first_Name,last_Name: first_Name+ " " + last_Name
print(full_Name("Bro","Code")

age_check=lambda age:True if age>=18 else False
print(age_check(12))
```

//

55. Sort

****

 # sort() method = used with lists
 # sort() function = used with iterables

```
students=["Squidward","Sandy","Patrick","Spomgebob","Mr.Krabs"] // this is a list

students.sort()
for i in students:
print(i)
```

if you use:

```
students.sort(reverse=True) //It will print but not alphabetical rather opposite but you need to have  [ ]
```

if you use ( ) for the list instead of [ ] then:

sorted_students = sorted(students) // and to reverse this then (students, reverse=True)

if you have tuple(keyword sorting will be used to organize by grade or name etc):

```
students =[
("Squidward","F",60),
("Sandy","A",33),
("Spongebob","B",20),
("Mr.Krabs","C",78)]
```

OBS.it is like columns like the name, the grade, age

```
students.sort() // will print alphabetically by the name
for i in students

grade=lambda grades:grades[1] // arrange by grades
students(key=grade) // if you need opposite then key=grade, reverse=True
for i in students:
print(i)

age=lambda ages:ages[2] // arrange by age
students(key=age) // if you need opposite then key=grade, reverse=True
for i in students:
print(i)
```

if there is a tuple of tuple, so instead of [( )] you have (( ))

```
students =(
("Squidward","F",60),
("Sandy","A",33),
("Spongebob","B",20),
("Mr.Krabs","C",78))

age=lambda ages:ages[2]
sorted_students=sorted(students,key=age)
for i in sorted_students:
print(i)
```

Caz.So this is how you sort an iterabile including a list

//

56. Map

****

 # map() = applies a function to each item in an iterable (list, tuple, etc.)
 # map(function, iterable) // so our map function accepts 2 arguments

```
store=[("shirt",20.00), // list of tuples named store
("pants",25.00),
("jackets",50.00),
("socks",10.00)]
```

*** So the prices above are in dollars and we want to convert them into euros, by using lambda function**

```
to_euros= lambda data: (data[0],data[1] * 0.82)
to_dollars= lambda data: (data[0],data[1] / 0.82)

store_euros = list (map(to_euros, store))

for i in store_euros:
print(i)
```

//

57. Filter

*****

 # filter() = creates collection of elements from an iterable for which a function returns true
 # filter(function, iterable)

```
friends[("Rachel",19), // list of tuples named friends
("Monica",18),
("Phoebe ",17),
("Joey",16),
("Chandler",21),
("Ross",20)]
```

*** So we want to create a separate list that are 18 and older, we use lambda**

```
age=lambda data:data[1] >= 18 // so basically it is a search result then we create a new list
drinking_bussies=list(filter(age, friends)) // cast to a new list and assign to a new list
for i in drinking_buddies:
print(i)
```

//

58. Reduce

******

 # reduce() = apply a function to an iterable and reduce it to a single cumulative value.
 # performs function on first two elements and repeats process until 1 value remains
 # reduce(function, iterable)

ex1:

*** let's imagine we are playing scrabble**

```
import functools // you have to import functools
letters=["H","E","L","L","O"] // this is a list
word=functools.reduce(lambda x,y,:x+y,letters)
print(word)
```

ex2:

*** let's say i want to find factorial of 5**

```
import functools // you have to import functools
factorial=[5,4,3,2,1]
result=functools.reduce(lambda x,y,:x*y,factorial) // so it will action on the first 2 then reult on the next and so on 5*4=20 then 20*3=60 then 60*2=120 & 120*1=120
print(result)
```

//

59. List Comprehension

****

 # List comprehension = a way to create a new list with less syntax
 # can mimic certain lambda functions, easier to read
 # list = [expression for item in iterable] << formula

 # list = [expression for item in iterable if conditional] << formula
 # list = [expression if/else for item in iterable] << formula

```
squares = []                # create an empty list
for i in range(1,11):       # create a for loop
squares.append(i * i )      # define what each loop iteration should do
print(squares)
```

or you can write it in fewer lines:

```
squares = [i * i for i in range(1,11)]
print(squares)
```

other example:

```
students = [100,90,80,70,60,50,40,30,0]
passed_students=list(filter(lamda x:x >= 60, students))
print(passed_students)
```

above can be writen as:

```
# passed_students=[i for i in students if i>=60]
 or # passed_students=[i if i>=60 else "Failed" for i in students]
print(passed_students)
```

//

60. Dictionary Comprehension

****

 # same as above just that they are for dictionaries
 # create dictionaries using an expression
 # can replace for loops and certain lambda functions

 # dictionary = {key: expression for (key, value) in iterable} << formula
 # dictionary = {key: expression for (key, value) in iterable if conditional} << formula
 # dictionary = {key: (if/else) for (key,value) in iterable} << formula
 # dictionary = {key: function(value) for (key,value) in iterable} << formula

```
cities _in_F = {'New York':32, 'Boston':75, 'Los Angeles':100, 'Chicago':50}

cities_in_C={key: round((value-32)*(5/9)) for (key,value) in cities_F.items()}
print(cities_in_C)
```

we can add also if:

```
weather = {'New York':"snwoing",'Boston':"sunny",'Los Angeles':"sunny",'Chicago':"cloudy"}
sunny_weather={key:value for (key, value) in weather.items() if value=="sunny"}  # << sunny_weather will be the name of our dictionary
print(sunny_weather)
```

3rd example we wil change the numbers to the fact if it is work or not:

```
cities = {'New York':32, 'Boston':75, 'Los Angeles':100, 'Chicago':50}
desc_cities= {key: ("Warm" if value >=40 else "Cold") for (key, value) in cities.items()}
print(desc_cities)
```

4th example to use function:

```
def check_temp(value): # define the function check_temp
if value >=70:
return "HOT"
elif 69>= value v>=40:
return "Warm"
else:
return "Cold"
cities = {'New York':32, 'Boston':75, 'Los Angeles':100, 'Chicago':50}
desc_cities= {key: check_temp(value) for (key, value) in cities.items()}
print(desc_cities)
```

//

61. Zip Function

****

 # zip(*iterable) = aggregate elements from two or more iterables (list, tuples, sets, etc,)
 # creates a zip object with paired elements stored in tuples for each element within our zip object

```
username=["Dude", "Bro", "Mister"] <<list of usernames
password=("p@ssword","abs123","guest")  << tuple of passwords

users = zip(usernames, passwords)
print(type(users) << to show that it is a zip object
for i in users:
print(i)
```

you can cast it:

```
users=list(zip(usernames, passwords))
users =dict(zip(usernames, passwords))
for key,value in users.items():
print(key+" : "+value)
```

you can use more than 2 iterables:

```
username=["Dude", "Bro", "Mister"] <<list of usernames
password=("p@ssword","abs123","guest")  << tuple of passwords
login_date=["1/1/2021","1/2/2021","1/3/2021"]
users=zip(usernames,passwords,login_date)
for i in users:
print(i)
```

//

62. if__name__='__main__'

****

 # y tho?
 # 1. Module can be run as a standalone program
 # 2. Module can be imported and used by other modules

 # python interpretor sets "special variables", one of which is __name__
 # then python will eexecute the code found within __main__

```
print(__name__) # it will print __main__
```

to import module from other module:

```
import module_two # module_two is the name of the file
print(__name__)
print(module_two.__name__) # will print __main__ and module_two
```

other example:

```
if__name__=='__main__':
print("running this module directly")
else:
print("running other module indirectly")
```

if I have a function in module 1 and try to use it in module 2 then:

```
import module_one
module_one.hello()
```

Obs.but you will not be able to run this function from module 1 anymore so you will have to write in module one:

```
def hello():
print("Hello") & add if __name__=='__main__': hello()
```

//

63. Time Module

****

 # epoch = a date and time from which a computer measures system time
 # epoch = when your computer thinks time began (reference point)

import time

 # Method 1
print(time.ctime(0)) # this will print reference time = **epoch = convert a time expressed in seconds since epoch to a readable string, so (0) is seconds**
 # so to change time/date if you pass (1000000) then time/date will change

 # Method 2
print(time.time()) **# return seconds since epoch**

 # Method 3
print(time.ctime(time.time())) **# will print current date and time**

 # Method 4
time_object=time.localtime() **# will create an object based on local time**
time_object = time.gmtime() **# if you want to use UTC or GMT**
print(time_object)
local_time=time.strftime("%B %d %Y %H:%M:%S", time_object) **# so % is actually directive**
print(local_time)

 # Method 5
time_string="20 April, 2020"
time_object=time.strp(time_string,"%d %B, %Y")
print(time_object)
 # (year, month,hours,secs,#day of the week, #day of the year, dst)

 # Method 6
time_tuple=(2020,4,20,4,20,0,0,0,0)
time_string=time.asctime(time_tuple)
print(time_string)

 # Method 7
time_tuple=(2020,4,20,4,20,0,0,0,0)
time_string=time.mktime(time_tuple)
print(time_string)

//

64. Multithreading

****

 # thread = a flow of execution. Like a separate order of instructions.
 # so we can have or program run its different parts at different times
 # however thread takes a turn running to achieve concurrency
 # GIL = (global interpreter lock), allows only one thread to hold the control of the python interpreter

 # cpu bound = program / task spends most of it's time waiting for internal events (CPU intensive) use multiprocessing

 # io bound = program/task spends most of it's time waiting for external events (user input, web scraping) use multithreading

import threading
import time
print(threading.active_count()**) << you can print number of active threads**
print(threading.enumerate())

example:
import threading
def eat_breakfast():
time.sleep(3)
print("you eat breakfast")

```
def drink_coffee():
time.sleep(4) # 4 seconds
print("you drank coffee")
def study():
time.sleep(5)
print("you finish study")

x = threading.Thread(target=eat_breakfast, args=()) # Thread 1 created
x.start()

y = threading.Thread(target=drink_coffee, args=()) # Thread 2
y.start()

z = threading.Thread(target=study, args=()) # Thread 3
z.start(

#eat_breakfast() // wil be part of main thread
#drink_coffee() // wil be part of main thread
#study() // wil be part of main thread

print(threading.active_count()) # main thread
print(threading.enumerate())
```

Obs.We wil have 4 threads and main one will be actioned first, so each task has been allocated to a thread so they will not run one after one rather in the same time

you can use:

```
print(time.perf()) # this will advise main thread to create those threads and then they will be actioned
```

you can also use:

```
x.join() # so main thread has to wait till thread x finish but when you reach the main thread then the x thread will be synchronized and not active
```

//

65. Daemon Threads

****

```
#daemon thread = a thread that runs in the background, not important for a program to run
# your program will not wait for daemon threads to complete before exiting
# non-daemon threads cannot normally be killed, stay alive until task is complete

# ex. background tasks, garbage collection, waiting for input, long running process

def timer():
print()
count=0
while True:
time.sleep(1)
count+=1
print("logged in for: ", count, "seconds")

x=threading.Thread(target-timer, daemon=True) # By adding daemon then the program will end this thread otherwise it will not even if main was executed
x.start()

answer = input("Do you wish to exit?")
```

obs. so our main thread will be in charge of running and waiting the input and the new thread will be running in the background counting time logged

you can also use:

```
x.setDaemon(True) # at the end
print(x.isDaemon()) # will test if the thread is deamon or not
```

Obs.So deamon thread is a thread that runs in the background but are not important for the program to keep running

//

66. Multiprocessing

****

```
# multiprocessing = running in parallel on different cpu cores, bypass GIL used for threding
# multiprocessing = better for cpu bound tasks (heavy cpu usage)
# multithreading = better for io bound  tasks (waiting around)

from multiprocessing import Process, cpu_count
import time

def counter(num):
count=0
while count <num:
count+=1

def main():
a = Process(target=counter, args =(10000000000,))
b = Process(target=counter, args =(10000000000,)) # by joining you speed up the process diving work load

a.start()
b.start()

a.join()
b.join()

print("finished in :", time.perf_counter(),"seconds")

if __name__ == '__main__': << you have to add ths if working in windows
main()
```

//

67. GUI Windows

****

```
from tkinter import *

# widgets = GIU elements: button, textboxes, labels, images
# windows = serves as a container to hold or contain these these widgets

window = Tk() # instantiate an instance of a window
window.geometry("420x420")
window.title("Bro Code first GUI program")

icon  = PhotoImage(file='logo.png')
window.iconphototo(True,icon)
window.config(background="black")
window.mainloop() # place window on computer screen, listen on events
```

//

68. Labels

******

```
# label = an area widget that holds text and/or an image within a window

from tkinter import *

windows = Tk()

photo = PhotoImage(file='C:\\Users\\Cakow\\Desktop\\person.png')

label = Label(window, text="Hello World", font = ('Arial',40,'bold'),fg='green', bg='black', relief='RAISED',bd=10,padx=20,pady=20,image=photo,compound='bottom') #fg will change font color, bg=background, relief=border,bd=border size
```

label.pack() or you can use label.place(x=0, y=0) **<< without it it won't show anything**

windows.mainloop()

//

69. Buttons
****

```
from tkinter import *
# button = you click it, then it does stuff

count = 0

def click():
global count << count how many you pressed on the button
count+=1
print(count)


window = Tk()

photo=PhotoImage(file='like.png')

button = Button(window,
text="click me!",
command=click, # you write this function without() and it is called callback
font=("Comic Sans,30"),
fg="#00FF00",
bg="black",
activeforeground="00FF00",
activebackground="black",
state=DISABLED, # so you will not be ablke to click on it or Active
image=photo,
compound='bottom' #or top)
button.pack()

window.mainloop()
```

//

70. Entry Box
****

```
from tkinter import *

# entry widget = textbox that accepts a single line of user input

def submit():
username=entry.get()
print("Helloo"+username)
entry.config(state=DISABLED) <<entry box will be disabled after submitting

def delete():
entry.delete(0,END)

def backspace():
entry.delete((entry.get())-1,END)

window = Tk()

entry = Entry(window,
font=("Arial",50),
```

```
fg="#00FF00",
bg="black",
show="*" # it will print * if we type password but the words will be stored correctly in the variable)
entry.insert(0, 'Spongebob')
entry.pack(side=LEFT)

submit_button = Button(window,text="submit", command=submit)
submit_button.pack(side=RIGHT)

delete_button = Button(window,text="delete", command=delete)
delete_button.pack(side=RIGHT)

backspace_button = Button(window,text="backspace", command=backspace)
backspace_button.pack(side=RIGHT)

Window.mainloop()
```

//

71. GUI Checkbox

****

```
from tkinter import *

def display():
if ((x.get)==1):
print("You agree")
else:
print("you don't agree")

window = Tk()

x= IntVar()

python_photo = PhotoImage(file='Python.png')

check_button = Checkbutton(window,
text="I agree to something",
variable = x,
onvalue=1,
offvalue=0,
command=display,
font=('Arial',20),
fg='#00FF00',
bg='black',
activeforeground='#00FF00',
activebackground='black',
padx=25,
pady=10,
image=python_photo,
compound='left')

check_button.pack()

window.mainloop()
```

//

72. GUI Radiobuttons

****

```
# radio button = similar to checkbox, but you can only select one from

from tkinter import *
```

```
food=["pizza", "hamburger","hotdog"]

def order():
if (x.get()==0):
print("you ordered pizza")
elif (x.get()==1):
print("you ordered a hamburger")
elif(x.get()==2):
print("you ordered a hotdog")
else:
print("huh?")

window = Tk()

pizzaImage = PhotoImage(file='pizza.png')
hamburgerImage = PhotoImage(file='phamburger.png')
hotdogImage = PhotoImage(file='hotdog.png')
foodImages = [pizzaImage,hamburgerImage, hotdogImage]

x=IntVar()

for index in range(len(food)):
radiobutton=Radiobutton(window,
text=food[index], # adds text to radio buttons
variable=x, # groups radiobuttons together if they share the same variable
value=index, # assigns each radiobutton a different value
padx=25, # adds padding on x-axis
font=("Impact",50),
image=foodImages[index], # adds image to radiobutton
compund = 'left', # adds image & text (left-side)
indicatoron=0, # elimiate circle indicators
width=375, # sets width of radio buttons
command = order # set command of radiobutton to function
)

radiobutton.pack(anchor=W)

window.mainloop()
```

//

73. GUI Scale

****

```
from tkinter import *

def submit():
print("The temperature is:" + str(scale.get())+ "degrees C")

window = Tk()

hotImage = PhotoImage(file='hot.png')
hotLabel = Label(image=hotImage)
hotLabel.pack()

scale = Scale(window,
from_=100,
to=0,
length=600,
orient=VERTICAL, #orientation of scale
font=('Consolas',20),
tickinterval=10, # adds numeric indicators for value
```

```
showvalue=0, # hide current value
resolution =5, # increment of slider
troughcolor='#69EAFF',
fg = '#FF1C00',
bg='#black'
)

scale.set(((scale['from']-scale['to'])/2+)+scale['to']) # make the slider go in the middle by using formula, not needed but just as add-on

scale.pack()

coldImage = PhotoImage(file='cold.png')
coldLabel = Label(image=coldImage)
coldLabel.pack()

button = Button(window, text='submit', command=submit)
button.pack()

window.mainloop()
```

//

74. GUI Listbox

****

```
# listbox = A listing of selectable text items within it's own container

def submit():
# print("you have ordered:")  <<for 1 selection
# print(listbox.get(listbox.curselection()))  <<for 1 selection
food=[ ]
for index in listbox.curselection():
food.insert(index,listbox.get(index))
print("You have ordered:")
for index in food:
print(index)

def add():
listbox.insert(listbox.size(),entryBox.get())
listbox.config(height=listbox.size()) # will adjust size dynamically

def delete():
# listbox.delete(listbox.curselection()) << for single selection
for index in reversed(listbox.curselection()):
listbox.delete(index)
listbox.config(height=listbox.size()) # will adjust size dynamically

from tkinter import *

window  = Tk()

listbox = Listbox(window,
bg="#f7ffde",
font=("Constantia",35),
width=12,
selectmode=MULTIPLE
)

listbox.pack()

listbox.insert(1,"pizza")
listbox.insert(2,"pasta")
listbox.insert(3,"garlic bread ")
```

```
listbox.insert(4,"soup")
listbox.insert(5,"salad")

listbox.config(height=listbox.size()) # will adjust size dynamically

entryBox= Entry(window)
entryBox.pack()

submitButton=Button(window,text="submit", command=submit)
submitButton.pack()

addButton=Button(window,text="add", command=add)
addButton.pack()

deleteButton=Button(window,text="delete", command=delete)
deleteButton.pack()

window.mainloop()
```

//

75. GUI MessageBox

****

```
from tkinter import *
from tkinter import messagebox # import messagebox library

def click():
# messagebox.showinfo(title='This is an info message box', message='You are a person')
# while(true): # to enter a loop with a message
# messagebox.showwarning(title='warning', message='You have a Virus')
# messagebox.showerror(title='error', message='something went wrong')

# if messagebox.askcancel(title='ask ok cancel', message='Do you want to do the thing?'):
# print('you did a thing')
# else:
# print('you caceled a thing')

# if messagebox.askrecancel(title='ask ok cancel', message='Do you want to retry the thing?'):
# print('you retried a thing')
# else:
# print('you caceled a thing')

# if messagebox.askyesno(title='ask yes or no', message='Do you like cake?):
# print('I like cake too')
# else:
# print('why do you not like cake?')

# answer = messagebox.askquestion(title='ask question',message='do you like pie?')
# if (answer =='yes'):
# print('i like pie too')
# else:
# print('why do you not like pie?')

answer=messagebox.askyesnocancel(title='yes no cancel', message='do you like to code?', icon='warning') # for icon you can use also 'info' or 'error'
if (answer==True):
print('you like to code')
elif(answer==False):
print('Then why are you watching video on coding?')
else:
print('you have dodged the question')
```

```
window = Tk()

button = Button(window, command=click,text='click me')
button.pack()

window.mainloop()
```

//

76. GUI Colorchooser

****

```
from tkinter import *
from tkinter import colorchooser # submodule

def click():
color = colorchooser.askcolor()
# print(color)
colorHex=color[1]
# print(colorHex)
window.config(bg=colorHex) # will change background color
or you can write the above in 2 lines as:
color=colorchooser.askcolor()
window.config(bg=color[1])
or you can write the above in one line as:
window.config(bg=colorchooser.askcolor()[1])

window = Tk()

window.geometry("420x420")
button=Button(text='click me', command=click )
button.pack()

window.mainloop()
```

//

77. GUI Text Area

****

```
# text widget = functions like a text area, you can enter multiple lines of text

def submit():
input = text.get("1.0",END)
print(input)

from tkinter import *

window=Th()

text=Text(window,
bg="light yellow",
font=("Ink Free",25),
height=8,
width=20,
padx=20,
pady=20,
fg="purple")
text.pack()

button=Button(window,text="submit",command=submit)
button.pack()

window.mainloop()
```

//

78. GUI Open A File (file dialog)

****

```
from tkinter import *
from tkinter import filedialog

def openFile():
filepath = filedialog.askopenfilename(initialdir="C:\\Users\\Cakow\\PycharmProjects\\Main", title="Open file okay",filetypes=("text files","*.txt"),("all files","*.*")))
# print(filepath)
file = open(filepath,'r')
print(file.read())
file.close()

window=Tk()

button = Button(text="Open", command=openFile)
button.pack()

window.mainloop()
```

//

79. GUI Save A File (File dialog)

****

```
from tkinter import *
from tkinter import filedialog

def saveFile():
file = filedialog.asksaveasfile(initialdir="C:\\Users\\Cakow\\PycharmProjects\\Main",
defaultextension='.txt',
filetypes=[("Text file",".txt"),
("HTML file",".html"),
("All file",".*"),
])
if file is None: # if you clos the window before saving with a name it will give an error
return
# filetext = str(text.get(1.0,END)) << using text area for input
filetext=input("Enter some text") << use console window for input
file.write(filetext)
file.close()


window = Tk()

button = Button(text='save', command=saveFile)
button.pack()
text=Text(window)
text.pack()

window.mainloop()
```

//

80. GUI Menubar

****

```
from tkinter import *

def openFile():
print("File has been opened")
```

```
def saveFile():
print("File has been Saved")

def cut():
print("you cut some text")

def copy():
print("you copied some text")

def paste():
print("you pasted some text")

window=Tk()

openImage = PhotoImage(file="file.png")
saveImage = PhotoImage(file="save.png")
exitImage = PhotoImage(file="exit.png")

menubar=Menu(window)

window.config(menu=menubar)

fileMenu = Menu(menubar, tearoff=0, font=("MV Boli",15))
menubar.add_cascade(label="File",menu=fileMenu)
fileMenu.add_command(label="Open", command=openFile, image=openImage,compound='left')
fileMenu.add_command(label="Save",command=saveFile,image=saveImage,compound='left')
fileMenu.add_seperator()
fileMenu.add_command(label="Exit",command=quit,image=exitImage,compound='left')

editMenu=Menu(menubar, tearoff=0,font=("MV Boli",15))
menubar.add_cascade(label="Edit",menu=editMenu)
editMenu.add_command(label="Copy", command=copy)
editMenu.add_command(label="Paste", command=paste)

window.mainloop()
```

//

81. GUI Frames

****

```
# frame = a rectangular container to group and hold widgets

from tkinter import *

window =Tk()

button = Button(window,text="W",font=("Consolas",25),width=3)
button.pack()

frame = Frame(window, bg="pink",bd=5,relief=SUNKEN)
# frame.pack(side=BOTTOM)
```

or you can use place to place the frame at specific coordinates:

```
frame.place(x=100,y=100)
```

or you can write above as:

```
Button(frame,text="W",font=("Consolas",25),width=3).pack(side=TOP)
Button(frame,text="A",font=("Consolas",25),width=3).pack(side=LEFT)
Button(frame,text="S",font=("Consolas",25),width=3).pack(side=LEFT)
Button(frame,text="D",font=("Consolas",25),width=3).pack(side=LEFT)

window.mainloop()
```

//

82. New Window

****

```
from tkinter import *

def create_window():
# new_window = Toplevel() # Toplevel() = new window 'on top' of other windows, linked to a 'bottom' window-if you close main window then the new window will close
# new_window = Tk() # TK () = new independent window
old_window.destroy() # close out of old window

old_window = Tk()

Button(old_window, text="create new window",command=create_window).pack()

old_window.mainloop()
```

//

83. Window Tabs

****

```
from tkinter import*
from tkinter import ttk

window=Tk()

notebook=ttk.Notebook(window) # widget that manages a collection of windows /displays

tab1 = Frame(notebook) # new frame for tab 1
tab2 = Frame(notebook) # new frame for tab 2

notebook.add(tab1,text="Tab1")
notebook.add(tab2,text="Tab2")
notebook.pack(expand=True,fill="both") # expand = expand to fill any space not otherwise used
                                       # fill = fill space on x and y axis (so the tabs will stick on top lft but you can expand the window with label)

Label(tab1,text="Hello, this is tab#1", width=50,height=25).pack()
Label(tab2,text="Goodbye, this is tab#2", width=50,height=25).pack()

window.mainloop()
```

//

84. GUI Grid

****

```
# grid() = geometry that organizes widgets in a table-like structure in a parent(like excel-column and rows)

from tkinter import *

window=Tk()

titleLabel  = Label(window,text="Enter your info",font=("Arial",25)).grid(row=0, column=0, columnspan=2)

firstNameLabel= Label(window, text="First name: ", width=20,bg="red").grid(row=1,column=0)
firstNameEntry = Entry(window).grid(row=1,column=1)

lastNameLabel= Label(window, text="Last name: ",bg="green").grid(row=2,column=0)
lastNameEntry = Entry(window).grid(row=2,column=1)

emailNameLabel= Label(window, text="email: ",bg="blue").grid(row=3,column=0)
emailNameEntry = Entry(window).grid(row=3,column=1)
```

```
submitButton=Button(window,text="Submit").grid(row=3,column=0,columnspan=2) # columnspan will take 2 columns (in between)

window.mainloop()
```

//

85. GUI Progress Bar

****

```
# GUI = Graphical User Interface

from tkinter import *
from tkinter.ttk import *
import time

def start():
GB=100
download = 0
speed=1
while (download<GB):
time.sleep(0.05)
bar['value']+=(speed/GB)*100
download+=speed
percent.set(str(int((download/GB)*100))+"%")
text.set(str(download)+"/"+str(GB)+" GB completed")
window.update_idletasks()

window = Tk()

percent=StringVar()
text = StringVar()

bar = Progressbar(window, orientation=HORIZONTAL, length=300)
bar.pack(pady=10)

percentLabel=Label(window,textvariable=percent).pack()
ptaskLabel=Label(window,textvariable=text).pack()

button = Button(window, text="download",command=start).pack()

window.mainloop()
```

//

86. GUI Canvas

****

```
# canvas = widget that is used to draw graphs, plots, images in a window

from tkinter import *

window=Tk()

# canvas=Canvas(window, height=500,width=500)
# canvas.create_line(0,0,500,500,fill="blue", width=5)
# canvas.create_line(0,500,500,0,fill="red", width=5)
# canvas.create_rectangle(50,50,250,250,fill="purple")
# points=[250,0,500,500,0,500]
# canvas.create_polygon(points,fill="yellow",outline="black",width=5)
# canvas.create_arc(0,0,500,500, fill="green", style=PIESLICE,start=180, extent=180)

create pokemon ball:
canvas.create_arc(0,0,500,500, fill="red", extent=180,width=10)
canvas.create_arc(0,0,500,500, fill="white", extent=180,start=180,width=10)
```

```
canvas.create_oval(190,190,310,310,fill="white",width=10)

canvas.pack()

window.mainloop()
```

//

87. GUI Keyboard Events

****

```
from tkinter import *

def doSomething(event):
#print("You pressed" + event.keysym) # will display what key was pressed
label.config(text=event.keysym)

window = Tk()

window.bind("<Key>",doSomething) # This will take an (event,function), retuen actually is the button enter, Key will type any key

label=Label(window,font="Helvetica",100))
label.pack()

window.mainloop()
```

//

88. GUI Mouse Events

****

```
from tkinter import *

def doSomething(event):
print("Mouse Coordinates" +str(event.x)+","+str(event.y))

window = Tk()

# window.bind("<Button-1>",doSomething)# will take 2 arguments (event,function) # left mouse click
# window.bind("<Button-2>",doSomething)# will take 2 arguments (event,function) # scroll wheel click
# window.bind("<Button-3>",doSomething)# will take 2 arguments (event,function) # right mouse click
# window.bind("<ButtonRelease>",doSomething) # will take 2 arguments (event,function)
# window.bind("<Enter>",doSomething)# will take 2 arguments (event,function) # it will activate function once entered in the window
# window.bind("<Leave>",doSomething)# will take 2 arguments (event,function) # it gives the coordinates once you leave the window
# window.bind("<Motion>",doSomething) # will take 2 arguments (event,function) #where the mouse moved - good for playing

window.mainloop()
```

//

89. Drag & Drop GUI

****

```
from tkinter import *

def drag_start(event):
widget = event.widget
widget.startX=event.x
widget.startY=event.y

def drag_motion(event):
widget = event.widget
x = widget.winfo_x() - widget.startX + event.x
y = widget.winfo_y() - widget.startY + event.y
widget.place(x=x,y=y)
```

```
window = Tk()

label=Label(window,bg="red",width=10,height=5)
label.place(x=0,y=0)

label=Label(window,bg="blue",width=10,height=5)
label.place(x=100,y=100)

label.bind("<Button-1>",drag_start)
label.bind("<B1-Motion>",drag_motion)


label2.bind("<Button-1>",drag_start)
label2.bind("<B1-Motion>",drag_motion)

window.mainloop()
```

//

90. Move Images w/keys

****

Without Canvas:

==========

```
from tkinter import *

def move_up(event):
label.place(x=label.winfo_x(), y=label.winfo_y()-10)

def move_down(event):
label.place(x=label.winfo_x(), y=label.winfo_y()+10)

def move_left(event):
label.place(x=label.winfo_x()-10, y=label.winfo_y())

def move_right(event):
label.place(x=label.winfo_x()+10, y=label.winfo_y())

window = Tk()
window.geometry("500x500")

window.bind("<w>",move_up)
window.bind("<s>",move_down)
window.bind("<a>",move_left)
window.bind("<d>",move_right)
window.bind("<Up>",move_up)
window.bind("<Down>",move_down)
window.bind("<Left>",move_left)
window.bind("<Right>",move_right)

myimage= PhotoImage(file='racingcar.pmg')
label=Label(window,image=myimage)
label.place(x=0,y=0)

window.mainloop()
```

With Canvas:

==========

```
from tkinter import *
```

```
def move_up(event):
canvas.move(myimage,0,-10)
def move_down(event):
canvas.move(myimage,0,10)
def move_left(event):
canvas.move(myimage,-10,0)
def move_right(event):
canvas.move(myimage,10,0)


window = Tk()

window.bind("<w>", move_up)
window.bind("<s>", move_down)
window.bind("<a>", move_left)
window.bind("<d>", move_right)
window.bind("<Up>", move_up)
window.bind("<Down>", move_down)
window.bind("<Left>", move_left)
window.bind("<Right>", move_right)

canvas = Canvas(window, width=500,height=500)
canvas.pack()

photoimage = PhotoImage(file='racecar.pmg')
myimage = canvas.create_image(0,0,image=photoimage,anchor=NW)

window.mainloop()
```

//

91. Animations

****

```
from tkinter import *
import time

WIDTH = 500 # it is with capital as it is a constant = it will nt be changed
HEIGHT =500
xVelocity = 3
yVelocity = 2

window=Tk()

canvas = Canvas(window,width=WIDTH, height=HEIGHT)
canvas.pack()

background_photo = PgotoImage(file='space.png') # create background before you add other images otherwise your background will overlap your other images
background=canvas.create_image(0,0,image=background_photo,anchor=NW)

photo_image = PgotoImage(file='ufo.png')
my_image=canvas.create_image(0,0,image=photo_image,anchor=NW)

image_width = photo_image.width()
image_height=photo_image.height()

while True:
coordinates = canvas.coords(my_image) # get coordinates where our image is located
print(coordinates) # print coordinates to the console window
if(coordinates[0]>=(WIDTH -image_width) or coordinates[0]<0): # so [0] is index 0 # so this will bounce the moving object to the other direction and before reaching the end of the window
xVelocity =  -xVelocity
if(coordinates[1]>=(HEIGHT -image_height) or coordinates[1]<0): # so [1] is index 1 # so this will bounce the moving object to the other direction and before reaching the end of the window
yVelocity =  -yVelocity
```

```
canvas.move(my_image,xVelocity,yVelocity) # move function takes 3 arguments (what you want to move, how far I want to move on each axis
window.update() # will update this window for any changes
time.sleep(0.01) # our thread that is in charge of running hte program will go to sleep for 0.01sec

window.mainloop()
```

//

92. Multiple Animations

****

```
from tkinter import *
import time
from Ball import *

window = Tk()

WIDTH = 500 # constant
HEIGHT = 500


canvas = Canvas(window, width=WIDTH,height=HEIGHT)
canvas.pack()

volley_ball=Ball(canvas,0,0,100,1,1,"white")
tennis_ball=Ball(canvas,0,0,50,4,3,"yellow")
basket_ball=Ball(canvas,0,0,125,8,7,"orange")

while True:
volley_ball.move()
tennis_ball.move()
basket_ball.move()

window.update()
time.sleep(0.01)

window.mainloop()
```

Ball.py

=====

```
class Ball: #This is the constructor
def __init__(self, canvas,x,y,diameter,xVelocity,yVelocity,color):
self.canvas=canvas
self.image=canvas.create_oval(x,y,diameter,diameter,fill=color)
self.xVelocity=xVelocity
self.yVelocity=yVelocity

def move(self):
coordinates=self.canvas.coords(self.image)
print(coordinates)
if(coordinates[2]>=(self.canvas,winfo_width()) or coordinates[0]<0):
self.xVelocity=-self.xVelocity
if(coordinates[3]>=(self.canvas,winfo_height()) or coordinates[1]<0):
self.yVelocity=-self.yVelocity
self.canvas.move(self.image,self.xVelocity,self.yVelocity)
```

//

93. Clock program

****

```
from tkinter import *
from time import *
```

```
def update():
    time_string = strftime("%I:%M:%S %p")
    time_label.config(text=time_string)

    day_string = strftime("%A")
    day_label.config(text=day_string)

    date_string = strftime("%B %d, %Y")
    date_label.config(text=date_string)

    window.after(1000,update)


window = Tk()

time_label = Label(window,font=("Arial",50),fg="#00FF00",bg="black")
time_label.pack()

day_label = Label(window,font=("Ink Free",25,"bold"))
day_label.pack()

date_label = Label(window,font=("Ink Free",30))
date_label.pack()

update()

window.mainloop()
```

//

94. Send An Email

****

```
import smtplib

sender="sender@gmail.com"
receiver= "receiver@gmail.com"
password="password123"
subject="Python email test"
body="I wrote an email"

#header
message = f"""From: Snoop Dogg{sender}
to: Nicholas Cage{receiver}
Subject: {subject}\n
{body}
"""
server=smtplib.SMTP("smtp.gmail.com",587) # 587 is the default mail submission port
server.starttls()

try:
server.login(sender,password)
print("Logged in..")
server.sendmail(sender, receiver, message)
print("email has been sent")
except smtplib.SMTPAuthenticationError:
print("unable to sign in")
```

//

95. Run With Command Prompt

****

 # run .py with cmd
 # save file as .py (Python file)
 # go to command prompt (for windows) or terminal (for Mac)
 # navigate to directory x/ your file: C:Users\BroCode\Desktop
 # invoke python interpreter + script: python hello_world.py

```
print("Hello World")
name=input("What's your name")
print("Hello "+name)
```

Obs.You invoke python interpreter as: python hello_world.py

//

96. pip

***

 # pip = package manager for packages and modules from python Package Index
 # included for python versions 3.4+:
 # help, check, update, installed packages, check outdated packages, install a package
 # pip: --version, install --upgrade pip, list, list --outdated, install "package name"

If you do not have latest python > go to sit > download > custom > optional features > next > click as python to environment variables > install
to use pip > open cmd prompt > type pip + enter
checking current version of pip = cmd prompt > type pip --version and enter
update pip > cmd prompt > type pip install --upgrade pip and enter
to get list of installed packages = > cmd prompt > pip list and enter
to see if some of the packages of pip are outdated = cmd prompt > pip list --outdated and enter
to upgrade one of the packages of pip = cmd prompt > pip install pygame --upgrade and enter
to install a package = cmd prompt > pip install pandas then enter
to check if package installed = cmd prompt > pip list and enter

Cz.write pip install
now we can import modules from this package like we import built in modules from python
to see what you installed with pip you go to pycharm -> external libraries -> Python 3.1 -> site-packages

Obs.For more packages go to Python packages index @pypi.org and make a search, example openpyxl you search in pypi.org

//

97. Py to Exe

****

 # windows Defender may prevent you from running
 # make sure pip and pyinstaller are installed/updated
 # F (all in 1 file)
#w (removes terminal window)
 # i icon.ico (adds custom icon to .exe)
 # clock.py (name of your main pythin file)
 # to convert file to .ico >> go to icoconvert.com
create a new folder
copy and relevent py folder or images in this folder
if you want an image for your executable also put it in the folder
open cmd prompt
change to directory of our new created python folder
so we need to convert python file to executable
type in cmd promot pyinstaller -F -w -i icon.ico clock.py # so convert in one file(F), do not use later the terminal (w), add icon (i) and at the end name of your python file
executable will be placed in a dist folder
drag and drop it where you want it to be

//

98. calculator program

```
    ***************
from tkinter import *
def button_press(num):
    global equation_text
    equation_text = equation_text + str(num)
    equation_label.set(equation_text)
def equals():
    global equation_text
    try:
        total = str(eval(equation_text))
        equation_label.set(total)
        equation_text = total
    except SyntaxError:
        equation_label.set("syntax error")
        equation_text = ""
    except ZeroDivisionError:
        equation_label.set("arithmetic error")
        equation_text = ""
def clear():
    global equation_text
    equation_label.set("")
    equation_text = ""


window = Tk()
window.title("Calculator program")
window.geometry("500x500")

equation_text = ""

equation_label = StringVar()

label = Label(window, textvariable=equation_label, font=('consolas',20), bg="white", width=24, height=2)
label.pack()

frame = Frame(window)
frame.pack()

button1 = Button(frame, text=1, height=4, width=9, font=35,
                 command=lambda: button_press(1))
button1.grid(row=0, column=0)
```

```
button2 = Button(frame, text=2, height=4, width=9, font=35,
            command=lambda: button_press(2))
button2.grid(row=0, column=1)

button3 = Button(frame, text=3, height=4, width=9, font=35,
            command=lambda: button_press(3))
button3.grid(row=0, column=2)

button4 = Button(frame, text=4, height=4, width=9, font=35,
            command=lambda: button_press(4))
button4.grid(row=1, column=0)

button5 = Button(frame, text=5, height=4, width=9, font=35,
            command=lambda: button_press(5))
button5.grid(row=1, column=1)

button6 = Button(frame, text=6, height=4, width=9, font=35,
            command=lambda: button_press(6))
button6.grid(row=1, column=2)

button7 = Button(frame, text=7, height=4, width=9, font=35,
            command=lambda: button_press(7))
button7.grid(row=2, column=0)

button8 = Button(frame, text=8, height=4, width=9, font=35,
            command=lambda: button_press(8))
button8.grid(row=2, column=1)

button9 = Button(frame, text=9, height=4, width=9, font=35,
            command=lambda: button_press(9))
button9.grid(row=2, column=2)

button0 = Button(frame, text=0, height=4, width=9, font=35,
            command=lambda: button_press(0))
button0.grid(row=3, column=0)

plus = Button(frame, text='+', height=4, width=9, font=35,
            command=lambda: button_press('+'))
plus.grid(row=0, column=3)

minus = Button(frame, text='-', height=4, width=9, font=35,
            command=lambda: button_press('-'))
minus.grid(row=1, column=3)

multiply = Button(frame, text='*', height=4, width=9, font=35,
            command=lambda: button_press('*'))
multiply.grid(row=2, column=3)

divide = Button(frame, text='/', height=4, width=9, font=35,
            command=lambda: button_press('/'))
divide.grid(row=3, column=3)

equal = Button(frame, text='=', height=4, width=9, font=35,
            command=equals)
equal.grid(row=3, column=2)

decimal = Button(frame, text='.', height=4, width=9, font=35,
            command=lambda: button_press('.'))
decimal.grid(row=3, column=1)

clear = Button(window, text='clear', height=4, width=12, font=35,
```

```
            command=clear)
clear.pack()

window.mainloop()
```

//

99. text editor program

```
    *****************
import os
from tkinter import *
from tkinter import filedialog, colorchooser, font
from tkinter.messagebox import *
from tkinter.filedialog import *


def change_color():
   color = colorchooser.askcolor(title="pick a color...or else")
   text_area.config(fg=color[1])


def change_font(*args):
   text_area.config(font=(font_name.get(), size_box.get()))


def new_file():
   window.title("Untitled")
   text_area.delete(1.0, END)


def open_file():
   file = askopenfilename(defaultextension=".txt",
                          file=[("All Files", "*.*"),
                                ("Text Documents", "*.txt")])

   if file is None:
      return

   else:
      try:
         window.title(os.path.basename(file))
         text_area.delete(1.0, END)

         file = open(file, "r")

         text_area.insert(1.0, file.read())

      except Exception:
         print("couldn't read file")

      finally:
         file.close()


def save_file():
   file = filedialog.asksaveasfilename(initialfile='unititled.txt',
                                       defaultextension=".txt",
                                       filetypes=[("All Files", "*.*"),
                                                  ("Text Documents", "*.txt")])

   if file is None:
      return
```

```
    else:
        try:
            window.title(os.path.basename(file))
            file = open(file, "w")

            file.write(text_area.get(1.0, END))

        except Exception:
            print("couldn't save file")

        finally:
            file.close()


def cut():
    text_area.event_generate("<<Cut>>")


def copy():
    text_area.event_generate("<<Copy>>")


def paste():
    text_area.event_generate("<<Paste>>")


def about():
    showinfo("About this program", "This is a program written by YOUUUUU!!!")


def quit():
    window.destroy()


window = Tk()
window.title("Text editor program")
file = None

window_width = 500
window_height = 500
screen_width = window.winfo_screenwidth()
screen_height = window.winfo_screenheight()

x = int((screen_width / 2) - (window_width / 2))
y = int((screen_height / 2) - (window_height / 2))

window.geometry("{}x{}+{}+{}".format(window_width, window_height, x, y))

font_name = StringVar(window)
font_name.set("Arial")

font_size = StringVar(window)
font_size.set("25")

text_area = Text(window, font=(font_name.get(), font_size.get()))

scroll_bar = Scrollbar(text_area)
window.grid_rowconfigure(0, weight=1)
window.grid_columnconfigure(0, weight=1)
text_area.grid(sticky=N + E + S + W)
scroll_bar.pack(side=RIGHT, fill=Y)
```

```
text_area.config(yscrollcommand=scroll_bar.set)

frame = Frame(window)
frame.grid()

color_button = Button(frame, text="color", command=change_color)
color_button.grid(row=0, column=0)

font_box = OptionMenu(frame, font_name, *font.families(), command=change_font)
font_box.grid(row=0, column=1)

size_box = Spinbox(frame, from_=1, to=100, textvariable=font_size, command=change_font)
size_box.grid(row=0, column=2)

menu_bar = Menu(window)
window.config(menu=menu_bar)

file_menu = Menu(menu_bar, tearoff=0)
menu_bar.add_cascade(label="File", menu=file_menu)
file_menu.add_command(label="New", command=new_file)
file_menu.add_command(label="Open", command=open_file)
file_menu.add_command(label="Save", command=save_file)
file_menu.add_separator()
file_menu.add_command(label="Exit", command=quit)

edit_menu = Menu(menu_bar, tearoff=0)
menu_bar.add_cascade(label="Edit", menu=edit_menu)
edit_menu.add_command(label="Cut", command=cut)
edit_menu.add_command(label="Copy", command=copy)
edit_menu.add_command(label="Paste", command=paste)

help_menu = Menu(menu_bar, tearoff=0)
menu_bar.add_cascade(label="Help", menu=help_menu)
help_menu.add_command(label="About", command=about)

window.mainloop()
```

//

100. tic tac toe game

****

```
from tkinter import *
import random

def next_turn(row, column):

    global player

    if buttons[row][column]['text'] == "" and check_winner() is False:

        if player == players[0]:

            buttons[row][column]['text'] = player

            if check_winner() is False:
                player = players[1]
                label.config(text=(players[1]+" turn"))

            elif check_winner() is True:
                label.config(text=(players[0]+" wins"))
```

```
            elif check_winner() == "Tie":
                label.config(text="Tie!")

        else:

            buttons[row][column]['text'] = player

            if check_winner() is False:
                player = players[0]
                label.config(text=(players[0]+" turn"))

            elif check_winner() is True:
                label.config(text=(players[1]+" wins"))

            elif check_winner() == "Tie":
                label.config(text="Tie!")

def check_winner():

    for row in range(3):
        if buttons[row][0]['text'] == buttons[row][1]['text'] == buttons[row][2]['text'] != "":
            buttons[row][0].config(bg="green")
            buttons[row][1].config(bg="green")
            buttons[row][2].config(bg="green")
            return True

    for column in range(3):
        if buttons[0][column]['text'] == buttons[1][column]['text'] == buttons[2][column]['text'] != "":
            buttons[0][column].config(bg="green")
            buttons[1][column].config(bg="green")
            buttons[2][column].config(bg="green")
            return True

    if buttons[0][0]['text'] == buttons[1][1]['text'] == buttons[2][2]['text'] != "":
        buttons[0][0].config(bg="green")
        buttons[1][1].config(bg="green")
        buttons[2][2].config(bg="green")
        return True

    elif buttons[0][2]['text'] == buttons[1][1]['text'] == buttons[2][0]['text'] != "":
        buttons[0][2].config(bg="green")
        buttons[1][1].config(bg="green")
        buttons[2][0].config(bg="green")
        return True

    elif empty_spaces() is False:

        for row in range(3):
            for column in range(3):
                buttons[row][column].config(bg="yellow")
        return "Tie"

    else:
        return False


def empty_spaces():

    spaces = 9

    for row in range(3):
        for column in range(3):
```

```
            if buttons[row][column]['text'] != "":
                spaces -= 1

    if spaces == 0:
        return False
    else:
        return True

def new_game():

    global player

    player = random.choice(players)

    label.config(text=player+" turn")

    for row in range(3):
        for column in range(3):
            buttons[row][column].config(text="",bg="#F0F0F0")


window = Tk()
window.title("Tic-Tac-Toe")
players = ["x","o"]
player = random.choice(players)
buttons = [[0,0,0],
           [0,0,0],
           [0,0,0]]

label = Label(text=player + " turn", font=('consolas',40))
label.pack(side="top")

reset_button = Button(text="restart", font=('consolas',20), command=new_game)
reset_button.pack(side="top")

frame = Frame(window)
frame.pack()

for row in range(3):
    for column in range(3):
        buttons[row][column] = Button(frame, text="",font=('consolas',40), width=5, height=2,
                              command= lambda row=row, column=column: next_turn(row,column))
        buttons[row][column].grid(row=row,column=column)

window.mainloop()
```

//

101. snake game

****

```
from tkinter import *
import random

GAME_WIDTH = 700
GAME_HEIGHT = 700
SPEED = 50
SPACE_SIZE = 50
BODY_PARTS = 3
SNAKE_COLOR = "#00FF00"
FOOD_COLOR = "#FF0000"
BACKGROUND_COLOR = "#000000"
```

```
class Snake:

    def __init__(self):
        self.body_size = BODY_PARTS
        self.coordinates = []
        self.squares = []

        for i in range(0, BODY_PARTS):
            self.coordinates.append([0, 0])

        for x, y in self.coordinates:
            square = canvas.create_rectangle(x, y, x + SPACE_SIZE, y + SPACE_SIZE, fill=SNAKE_COLOR, tag="snake")
            self.squares.append(square)


class Food:

    def __init__(self):

        x = random.randint(0, (GAME_WIDTH / SPACE_SIZE)-1) * SPACE_SIZE
        y = random.randint(0, (GAME_HEIGHT / SPACE_SIZE) - 1) * SPACE_SIZE

        self.coordinates = [x, y]

        canvas.create_oval(x, y, x + SPACE_SIZE, y + SPACE_SIZE, fill=FOOD_COLOR, tag="food")


def next_turn(snake, food):

    x, y = snake.coordinates[0]

    if direction == "up":
        y -= SPACE_SIZE
    elif direction == "down":
        y += SPACE_SIZE
    elif direction == "left":
        x -= SPACE_SIZE
    elif direction == "right":
        x += SPACE_SIZE

    snake.coordinates.insert(0, (x, y))

    square = canvas.create_rectangle(x, y, x + SPACE_SIZE, y + SPACE_SIZE, fill=SNAKE_COLOR)

    snake.squares.insert(0, square)

    if x == food.coordinates[0] and y == food.coordinates[1]:

        global score

        score += 1

        label.config(text="Score:{}".format(score))

        canvas.delete("food")

        food = Food()

    else:
```

```
            del snake.coordinates[-1]

            canvas.delete(snake.squares[-1])

            del snake.squares[-1]

        if check_collisions(snake):
            game_over()

        else:
            window.after(SPEED, next_turn, snake, food)


def change_direction(new_direction):

    global direction

    if new_direction == 'left':
        if direction != 'right':
            direction = new_direction
    elif new_direction == 'right':
        if direction != 'left':
            direction = new_direction
    elif new_direction == 'up':
        if direction != 'down':
            direction = new_direction
    elif new_direction == 'down':
        if direction != 'up':
            direction = new_direction


def check_collisions(snake):

    x, y = snake.coordinates[0]

    if x < 0 or x >= GAME_WIDTH:
        return True
    elif y < 0 or y >= GAME_HEIGHT:
        return True

    for body_part in snake.coordinates[1:]:
        if x == body_part[0] and y == body_part[1]:
            return True

    return False


def game_over():

    canvas.delete(ALL)
    canvas.create_text(canvas.winfo_width()/2, canvas.winfo_height()/2,
                       font=('consolas',70), text="GAME OVER", fill="red", tag="gameover")


window = Tk()
window.title("Snake game")
window.resizable(False, False)

score = 0
direction = 'down'

label = Label(window, text="Score:{}".format(score), font=('consolas', 40))
```

```
label.pack()

canvas = Canvas(window, bg=BACKGROUND_COLOR, height=GAME_HEIGHT, width=GAME_WIDTH)
canvas.pack()

window.update()

window_width = window.winfo_width()
window_height = window.winfo_height()
screen_width = window.winfo_screenwidth()
screen_height = window.winfo_screenheight()

x = int((screen_width/2) - (window_width/2))
y = int((screen_height/2) - (window_height/2))

window.geometry(f"{window_width}x{window_height}+{x}+{y}")

window.bind('<Left>', lambda event: change_direction('left'))
window.bind('<Right>', lambda event: change_direction('right'))
window.bind('<Up>', lambda event: change_direction('up'))
window.bind('<Down>', lambda event: change_direction('down'))

snake = Snake()
food = Food()

next_turn(snake, food)

window.mainloop()
```

//

102. Overloading Operators

****

```
class Point():
  def __init__(self, x=0, y=0):
          self.x=x
          self.y=y
          self.coords=(self.x, self.y)

  def move(self, x,y):
          self.x +=x
          self.y +=y

  def __add__(self,p):
          return Point(self.x +p.x, self.y +p.y)
                p1     p2    p1    p2
  def __str__(self):
          return "(" +str(self.x) + ',' +str(self.y) + ')' //(6,6)

p1=Point(3,4)
p2=Point(3,2)

p3=Point(1,3)
p4=Point(0,1)

p5=p1+p2
print(p5)
```

p1+p2 **# p1 becomes self and p2 becomes p**

5+6 **# 55& 6 are Operands and + is operator**

```
a=5
b='world'
print(a+b) # gives error
```

Obs. Python has synthetic sugar = already coded stuff to make it easier so 2+2=4 but if you put int with a string then you will have an error

print(int. __add__(a,b) is same as print(a+b**) -> add is a method of the int class -> so you are calling behind the scene the method add when you say a+b**

Magic Methodes:
__add__(self,other) +
__sub__(self,other) -
__mul__(self,other) *
__div__(self,other) /
__lt__(self,other) <
__gt__(self,other) >
__ge__(self,other) >=

```
class Students:
def __init__(self,m1,m2):
self.m1=m1
self.m2=m2

def __add__(self,other):
m1 = self.m1 + other.m1
m2 = self.m2 + other.m2

s3= Student(m1,m2)
return s3

def __gt__(self,other): #gt = greater than, ge = greater equal to
r1=self.m1+self.m2
r2=other.m1+m2
if r1 > r2:
True
else:
return False

def __str__(self):
return '() ()' .format(self.m1, self.m2)  58 69

s1=Student(58,69)
s2=Student(60,65)

s3=s1+s2 # error
```

but with overloading:
s3=s1+s2 **-> will be converted into Student. __add__(s1,s2) = s1 will become self and s2 becomes 'other'**

```
print(s3.m1) 3 118

if s1 > s2
print("s1 wins")
else:
print("s2 wins")

print(s1)
```

//

103. Method Overloading - you can do with 'this' or *args
****

 # if you have a class with 2 methods of same name then they will have different ar/par
ex.
Student:
def average(a,b)
def average(a,b,c)
#in python we do not have this possibility

```
class Student:
def __init__(self,m1,m2):
self.m1=m1
self.m2=m2

def sum(self,a,b)
a=a+b
return s

def sum(self,a=None,b=None,c=None): #we can declare none to all for it to work
s=0
if a!=None and b!=None and c!=None:#passing 3 arguments
s=a+b+c
elif a!=None and b!=None: #passing 2 arguments
s=a+b
else: #passing 1 argument
s=a

s1=Student(58,69)

print(s1.sum(5,9))
print(s1.sum(5,9,5))
```

//

104. Optional Parameters Tutorial #1
****

```
def fun(x=1):#so x will be overridden
return x **2

call=func(5)
print(call)

def func1(word, add=5,freq=1):
print(word*(freq+add))
call2=func1('hello',0)
print(call2)
```

example:

```
class car(object):
def __init__(self,make,model,year,condition='New',km=0):
self.make=make
self.model=model
self.year=year
self.condition=condition
self.kms=kms

def display(self, showAll=True):
if showAll:
print("This is a %s %s from %s, it is %s and has %s kms."%(self.make,self.model,self.year,self.condition,self.kms)
else:
print("This car is a %s %s from %s."%(self.model,self.model,self.year)

whip=car('Ford','Fusion',2012)
whip.display(False)
```

//

105. Private and Public Classes

****

file(mod.py)

```
class _Private: # _makes it private
def __init__(self,name):
self.name=name

class NotPrivate:
def __init__(self,name):
self.name=name
self.priv= _Private

def _display(self):#private method
print("hello")

def display(self):
print('hi')
```

file(tutorial.py)

```
import mod
from mod import NotPrivate

test=NotPrivate('tim')
test.display() // hi
test._display()//hello
```

//

106. Static Methods & Class Methods

****

```
class Dog:
dogs=[]

def __init(self,name):
self.name=name
self.dogs.append(self) # same as above just that this appends every single dog in the list

tim=Dog("Tim")
fim=Dog('Jim')
//classs variables you do it at the top of the class> Class Dogs:>dogs=[]<so not inside the methods
print(Dog.dogs) # so no need to create instance
```

//

107. Static Methods and Class

****

@=decorators->indicate special type of method

```
class Dog:
dogs=[]

def __init(self,name):
self.name=name
self.dogs.append(self) # same as above just that this appends every single dog in the list

@classmethod
def num_dogs(cls):
return len(cls.dogs)

@staticmethod
def bark(n):
"""barks n times"""
for _ in range(n):
print("Bark!")
```

print(Dog.num_dogs()) **# this will print without instantiate**
#you have to use cls instead of self

Dog.bark(5) **### this will work also**

//

108. Unpacking
****

➢ **unpacking:**
coordinates = (1, 2, 3) # works for list also [1, 2, 3]
x = coordinates[0]
y = coordinates[1]
z = coordinates[2]

➢ **or you can write above as:**
x,y,z = coordinates # packing
print(x) # unpacking

//

109. Packages
****

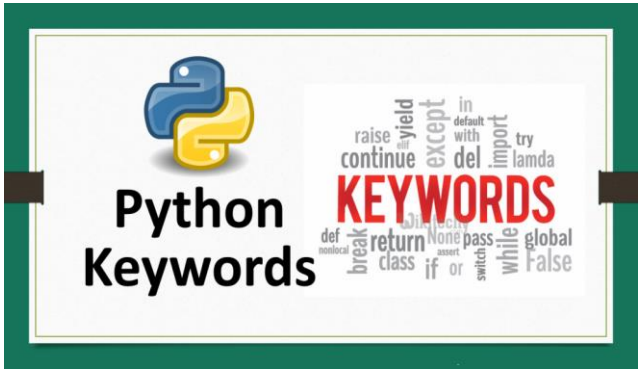
➢ **Package: container for multiple related modules, like in a mall we have Men's, women's, kid's clothing so that is like a package, at men's section we have shoes, t-shirt, and jackets so those are seen as modules**

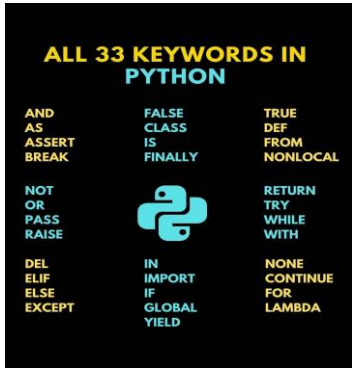
➢ **to create package: right click -> python file -> __init__**

➢ **or go to new and click python package that automatically will add __init__**

""" access a function from a module of a package """
from ecommerce.shipping import calc_shipping <- specific
calc_shipping()
OR
from ecommerce import shipping <- general
shipping.calc_shipping()

//

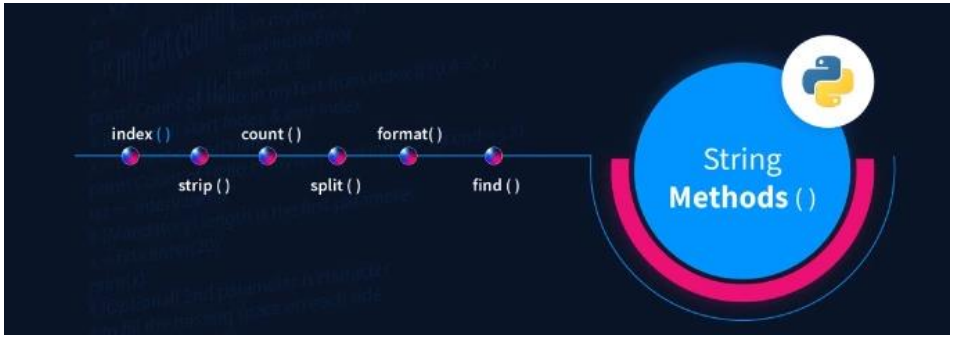




• **Python has a set of keywords that are reserved words that cannot be used as variable names, function names, or any other identifiers:**

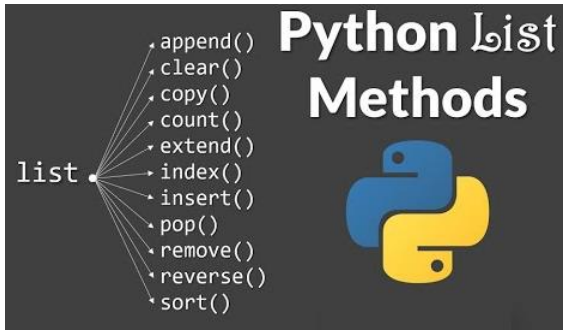
Keyword	Description
assert	For debugging

and	A logical operator
or	A logical operator
not	A logical operator
is	To test if two variables are equal
in	To check if a value is present in a list, tuple, etc.
None	Represents a null value
True	Boolean value, result of comparison operations
False	Boolean value, result of comparison operations
as	To create an alias
if	To make a conditional statement
elif	Used in conditional statements, same as else if
else	Used in conditional statements
while	To create a while loop
for	To create a for loop
return	To exit a function and return a value
continue	To continue to the next iteration of a loop
break	To break out of a loop
def	To define a function
pass	A null statement, a statement that will do nothing
class	To define a class
global	To declare a global variable
del	To delete an object
except	Used with exceptions, what to do when an exception occurs
finally	Used with exceptions, a block of code that will be executed no matter if there is an exception or not
from	To import specific parts of a module
import	To import a module
lambda	To create an anonymous function
nonlocal	To declare a non-local variable
raise	To raise an exception
try	To make a try...except statement
with	Used to simplify exception handling
yield	To end a function, returns a generator



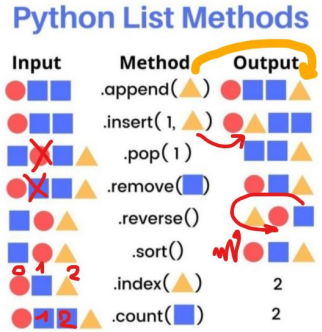
- Python has a set of built-in methods that you can use on strings.

Note: All string methods returns new values. They do not change the original string.

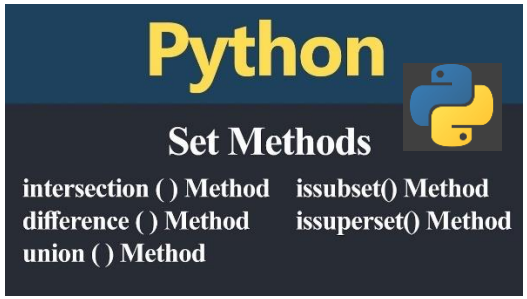


- Python has a set of built-in methods that you can use on lists/arrays

  Note: Python does not have built-in support for Arrays, but Python Lists can be used instead



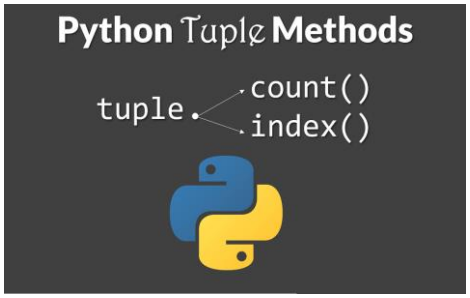
Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list



- Python has a set of built-in methods that you can use on sets.

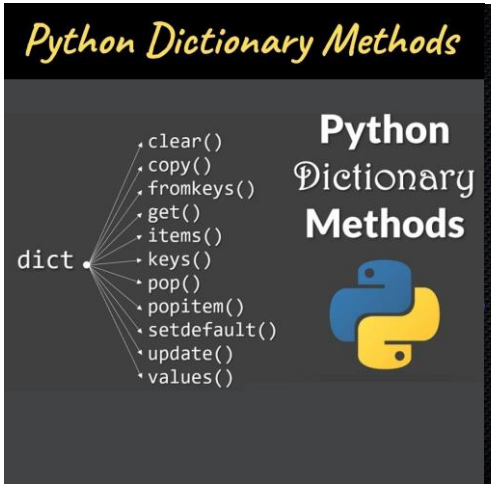
Method	Description

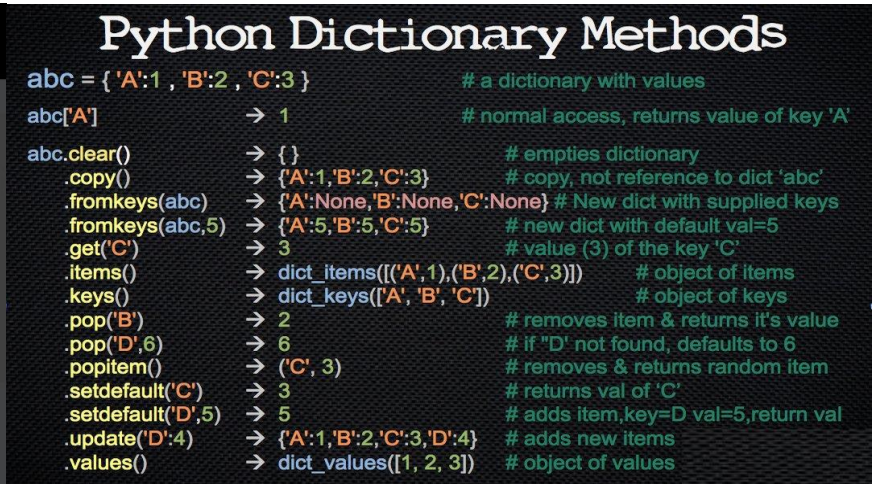
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two or more sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with another set, or any other iterable



- Python has two built-in methods that you can use on tuples

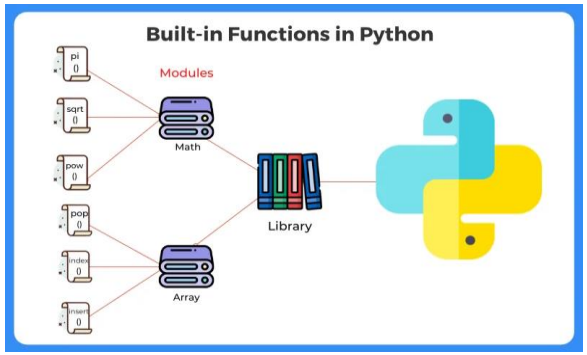
Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found





- Python has a set of built-in methods that you can use on dictionaries.

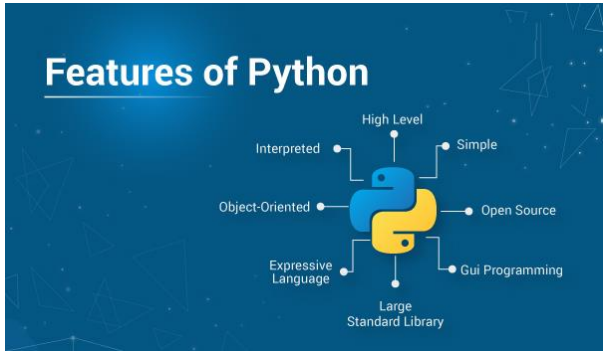
Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary



• Python has a set of built-in functions.

Function	Description
help()	Executes the built-in help system
id()	Returns the id of an object
type()	Returns the type of an object
len()	Returns the length of an object
list()	Returns a list
set()	Returns a new set object
tuple()	Returns a tuple
dict()	Returns a dictionary (Array)
format()	Formats a specified value
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
chr()	Returns a character from the specified Unicode code
str()	Returns a string object
sorted()	Returns a sorted list
slice()	Returns a slice object
int()	Returns an integer number
float()	Returns a floating point number
sum()	Sums the items of an iterator
any()	Returns True if any item in an iterable object is true
all()	Returns True if all items in an iterable object are true
hash()	Returns the hash value of a specified object
input()	Allowing user input
print()	Prints to the standard output device
abs()	Returns the absolute value of a number
memoryview()	Returns a memory view object
bin()	Returns the binary version of a number
bool()	Returns the Boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False

classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval()	Evaluates and executes an expression
exec()	Executes the specified code (or object)
filter()	Use a filter function to exclude items in an iterable object
frozenset()	Returns a frozenset object
getattr()	Returns the value of the specified attribute (property or method)
globals()	Returns the current global symbol table as a dictionary
hasattr()	Returns True if the specified object has the specified attribute (property/method)
hex()	Converts a number into a hexadecimal value
isinstance()	Returns True if a specified object is an instance of a specified object
issubclass()	Returns True if a specified class is a subclass of a specified object
iter()	Returns an iterator object
locals()	Returns an updated dictionary of the current local symbol table
map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable
min()	Returns the smallest item in an iterable
next()	Returns the next item in an iterable
object()	Returns a new object
oct()	Converts a number into an octal
open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
property()	Gets, sets, deletes a property
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator
round()	Rounds a numbers
setattr()	Sets an attribute (property/method) of an object
staticmethod()	Converts a method into a static method
super()	Returns an object that represents the parent class
vars()	Returns the __dict__ property of an object
zip()	Returns an iterator, from two or more iterators

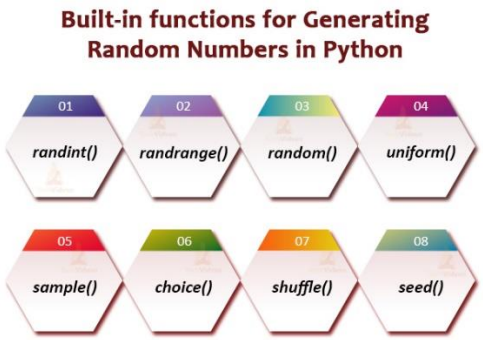


Feature	Description
Indentation	Indentation refers to the spaces at the beginning of a code line
Comments	Comments are code lines that will not be executed
Multi Line Comments	How to insert comments on multiple lines
Creating Variables	Variables are containers for storing data values
Variable Names	How to name your variables

Assign Values to Multiple Variables	How to assign values to multiple variables
Output Variables	Use the print statement to output variables
String Concatenation	How to combine strings
Global Variables	Global variables are variables that belongs to the global scope
Built-In Data Types	Python has a set of built-in data types
Getting Data Type	How to get the data type of an object
Setting Data Type	How to set the data type of an object
Numbers	There are three numeric types in Python
Int	The integer number type
Float	The floating number type
Complex	The complex number type
Type Conversion	How to convert from one number type to another
Random Number	How to create a random number
Specify a Variable Type	How to specify a certain data type for a variable
String Literals	How to create string literals
Assigning a String to a Variable	How to assign a string value to a variable
Multiline Strings	How to create a multi line string
Strings are Arrays	Strings in Python are arrays of bytes representing Unicode characters
Slicing a String	How to slice a string
Negative Indexing on a String	How to use negative indexing when accessing a string
String Length	How to get the length of a string
Check In String	How to check if a string contains a specified phrase
Format String	How to combine two strings
Escape Characters	How to use escape characters
Boolean Values	True or False
Evaluate Booleans	Evaluate a value or statement and return either True or False
Return Boolean Value	Functions that return a Boolean value
Operators	Use operator to perform operations in Python
Arithmetic Operators	Arithmetic operator are used to perform common mathematical operations
Assignment Operators	Assignment operators are use to assign values to variables
Comparison Operators	Comparison operators are used to compare two values
Logical Operators	Logical operators are used to combine conditional statements
Identity Operators	Identity operators are used to see if two objects are in fact the same object
Membership Operators	Membership operators are used to test is a sequence is present in an object
Bitwise Operators	Bitwise operators are used to compare (binary) numbers
Lists	A list is an ordered, and changeable, collection
Access List Items	How to access items in a list
Change List Item	How to change the value of a list item
Loop Through List Items	How to loop through the items in a list
List Comprehension	How use a list comprehensive
Check if List Item Exists	How to check if a specified item is present in a list
List Length	How to determine the length of a list
Add List Items	How to add items to a list
Remove List Items	How to remove list items
Copy a List	How to copy a list
Join Two Lists	How to join two lists
Tuple	A tuple is an ordered, and unchangeable, collection
Access Tuple Items	How to access items in a tuple
Change Tuple Item	How to change the value of a tuple item
Loop List Items	How to loop through the items in a tuple
Check if Tuple Item Exists	How to check if a specified item is present in a tuple
Tuple Length	How to determine the length of a tuple
Tuple With One Item	How to create a tuple with only one item
Remove Tuple Items	How to remove tuple items
Join Two Tuples	How to join two tuples
Set	A set is an unordered, and unchangeable, collection
Access Set Items	How to access items in a set
Add Set Items	How to add items to a set
Loop Set Items	How to loop through the items in a set
Check if Set Item Exists	How to check if a item exists
Set Length	How to determine the length of a set
Remove Set Items	How to remove set items

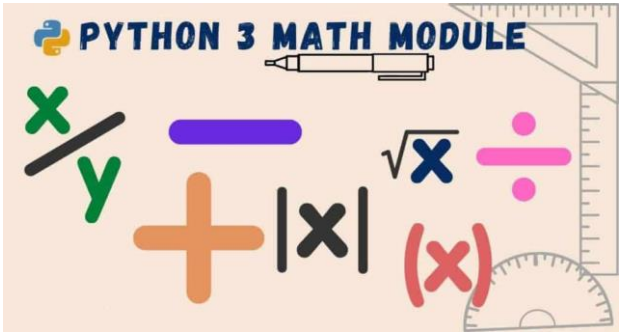
Join Two Sets	How to join two sets
Dictionary	A dictionary is an unordered, and changeable, collection
Access Dictionary Items	How to access items in a dictionary
Change Dictionary Item	How to change the value of a dictionary item
Loop Dictionary Items	How to loop through the items in a tuple
Check if Dictionary Item Exists	How to check if a specified item is present in a dictionary
Dictionary Length	How to determine the length of a dictionary
Add Dictionary Item	How to add an item to a dictionary
Remove Dictionary Items	How to remove dictionary items
Copy Dictionary	How to copy a dictionary
Nested Dictionaries	A dictionary within a dictionary
If Statement	How to write an if statement
If Indentation	If statements in Python relies on indentation (whitespace at the beginning of a line)
Elif	elif is the same as "else if" in other programming languages
Else	How to write an if...else statement
Shorthand If	How to write an if statement in one line
Shorthand If Else	How to write an if...else statement in one line
If AND	Use the and keyword to combine if statements
If OR	Use the or keyword to combine if statements
Nested If	How to write an if statement inside an if statement
The pass Keyword in If	Use the pass keyword inside empty if statements
While	How to write a while loop
While Break	How to break a while loop
While Continue	How to stop the current iteration and continue wit the next
While Else	How to use an else statement in a while loop
For	How to write a for loop
Loop Through a String	How to loop through a string
For Break	How to break a for loop
For Continue	How to stop the current iteration and continue wit the next
Looping Through a rangee	How to loop through a range of values
For Else	How to use an else statement in a for loop
Nested Loops	How to write a loop inside a loop
For pass	Use the pass keyword inside empty for loops
Function	How to create a function in Python
Call a Function	How to call a function in Python
Function Arguments	How to use arguments in a function
*args	To deal with an unknown number of arguments in a function, use the * symbol before the parameter name
Keyword Arguments	How to use keyword arguments in a function
**kwargs	To deal with an unknown number of keyword arguments in a function, use the * symbol before the parameter name
Default Parameter Value	How to use a default parameter value
Passing a List as an Argument	How to pass a list as an argument
Function Return Value	How to return a value from a function
The pass Statement i Functions	Use the pass statement in empty functions
Function Recursion	Functions that can call itself is called recursive functions
Lambda Function	How to create anonymous functions in Python
Why Use Lambda Functions	Learn when to use a lambda function or not
Array	Lists can be used as Arrays
What is an Array	Arrays are variables that can hold more than one value
Access Arrays	How to access array items
Array Length	How to get the length of an array
Looping Array Elements	How to loop through array elements
Add Array Element	How to add elements from an array
Remove Array Element	How to remove elements from an array
Array Methods	Python has a set of Array/Lists methods
Class	A class is like an object constructor
Create Class	How to create a class
The Class __init__() Function	The __init__() function is executed when the class is initiated
Object Methods	Methods in objects are functions that belongs to the object
self	The self parameter refers to the current instance of the class
Modify Object Properties	How to modify properties of an object
Delete Object Properties	How to modify properties of an object
Delete Object	How to delete an object

Class pass Statement	Use the pass statement in empty classes
Create Parent Class	How to create a parent class
Create Child Class	How to create a child class
Create the __init__() Function	How to create the __init__() function
super Function	The super() function make the child class inherit the parent class
Add Class Properties	How to add a property to a class
Add Class Methods	How to add a method to a class
Iterators	An iterator is an object that contains a countable number of values
Iterator vs Iterable	What is the difference between an iterator and an iterable
Loop Through an Iterator	How to loop through the elements of an iterator
Create an Iterator	How to create an iterator
StopIteration	How to stop an iterator
Global Scope	When does a variable belong to the global scope?
Global Keyword	The global keyword makes the variable global
Create a Module	How to create a module
Variables in Modules	How to use variables in a module
Renaming a Module	How to rename a module
Built-in Modules	How to import built-in modules
Using the dir() Function	List all variable names and function names in a module
Import From Module	How to import only parts from a module
Datetime Module	How to work with dates in Python
Date Output	How to output a date
Create a Date Object	How to create a date object
The strftime Method	How to format a date object into a readable string
Date Format Codes	The datetime module has a set of legal format codes
JSON	How to work with JSON in Python
Parse JSON	How to parse JSON code in Python
Convert into JSON	How to convert a Python object in to JSON
Format JSON	How to format JSON output with indentations and line breaks
Sort JSON	How to sort JSON
RegEx Module	How to import the regex module
RegEx Functions	The re module has a set of functions
Metacharacters in RegEx	Metacharacters are characters with a special meaning
RegEx Special Sequences	A backslash followed by a a character has a special meaning
RegEx Sets	A set is a set of characters inside a pair of square brackets with a special meaning
RegEx Match Object	The Match Object is an object containing information about the search and the result
Install PIP	How to install PIP
PIP Packages	How to download and install a package with PIP
PIP Remove Package	How to remove a package with PIP
Error Handling	How to handle errors in Python
Handle Many Exceptions	How to handle more than one exception
Try Else	How to use the else keyword in a try statement
Try Finally	How to use the finally keyword in a try statement
raise	How to raise an exception in Python



- Python has a built-in module that you can use to make random numbers.
  - The random module has a set of methods:

Method	Description
random()	Returns a random float number between 0 and 1
randrange()	Returns a random number between the given range
randint()	Returns a random number between the given range
choice()	Returns a random element from the given sequence
shuffle()	Takes a sequence and returns the sequence in a random order
seed()	Initialize the random number generator
getstate()	Returns the current internal state of the random number generator
setstate()	Restores the internal state of the random number generator
getrandbits()	Returns a number representing the random bits
choices()	Returns a list with a random selection from the given sequence
sample()	Returns a given sample of a sequence
uniform()	Returns a random float number between two given parameters
triangular()	Returns a random float number between two given parameters, you can also set a mode parameter to specify the midpoint between the two other parameters
betavariate()	Returns a random float number between 0 and 1 based on the Beta distribution (used in statistics)
expovariate()	Returns a random float number based on the Exponential distribution (used in statistics)
gammavariate()	Returns a random float number based on the Gamma distribution (used in statistics)
gauss()	Returns a random float number based on the Gaussian distribution (used in probability theories)
lognormvariate()	Returns a random float number based on a log-normal distribution (used in probability theories)
normalvariate()	Returns a random float number based on the normal distribution (used in probability theories)
vonmisesvariate()	Returns a random float number based on the von Mises distribution (used in directional statistics)
paretovariate()	Returns a random float number based on the Pareto distribution (used in probability theories)
weibullvariate()	Returns a random float number based on the Weibull distribution (used in statistics)



- Python has a built-in module that you can use for mathematical tasks
  - The math module has a set of methods and constants

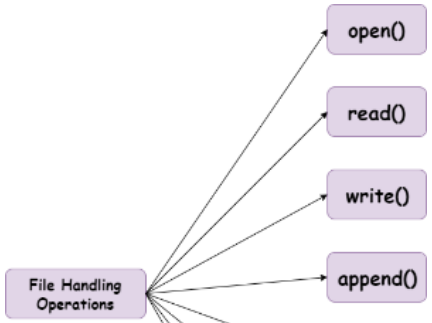
Method	Description
math.acos()	Returns the arc cosine of a number
math.acosh()	Returns the inverse hyperbolic cosine of a number
math.asin()	Returns the arc sine of a number
math.asinh()	Returns the inverse hyperbolic sine of a number
math.atan()	Returns the arc tangent of a number in radians
math.atan2()	Returns the arc tangent of y/x in radians
math.atanh()	Returns the inverse hyperbolic tangent of a number
math.ceil()	Rounds a number up to the nearest integer
math.comb()	Returns the number of ways to choose k items from n items without repetition and order
math.copysign()	Returns a float consisting of the value of the first parameter and the sign of the second parameter
math.cos()	Returns the cosine of a number
math.cosh()	Returns the hyperbolic cosine of a number
math.degrees()	Converts an angle from radians to degrees
math.dist()	Returns the Euclidean distance between two points (p and q), where p and q are the coordinates of that point

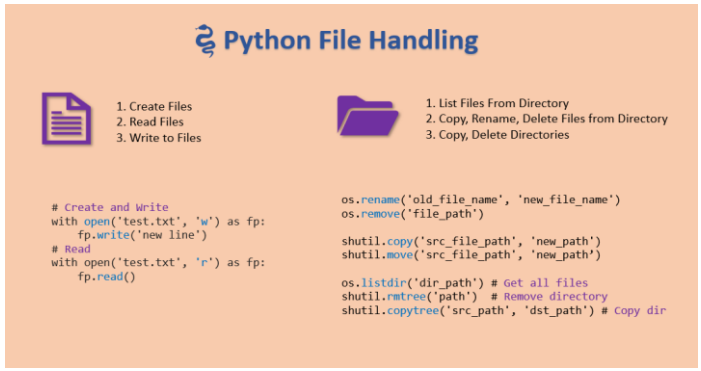
math.erf()	Returns the error function of a number
math.erfc()	Returns the complementary error function of a number
math.exp()	Returns E raised to the power of x
math.expm1()	Returns Ex - 1
math.fabs()	Returns the absolute value of a number
math.factorial()	Returns the factorial of a number
math.floor()	Rounds a number down to the nearest integer
math.fmod()	Returns the remainder of x/y
math.frexp()	Returns the mantissa and the exponent, of a specified number
math.fsum()	Returns the sum of all items in any iterable (tuples, arrays, lists, etc.)
math.gamma()	Returns the gamma function at x
math.gcd()	Returns the greatest common divisor of two integers
math.hypot()	Returns the Euclidean norm
math.isclose()	Checks whether two values are close to each other, or not
math.isfinite()	Checks whether a number is finite or not
math.isinf()	Checks whether a number is infinite or not
math.isnan()	Checks whether a value is NaN (not a number) or not
math.isqrt()	Rounds a square root number downwards to the nearest integer
math.ldexp()	Returns the inverse of math.frexp() which is x * (2**i) of the given numbers x and i
math.lgamma()	Returns the log gamma value of x
math.log()	Returns the natural logarithm of a number, or the logarithm of number to base
math.log10()	Returns the base-10 logarithm of x
math.log1p()	Returns the natural logarithm of 1+x
math.log2()	Returns the base-2 logarithm of x
math.perm()	Returns the number of ways to choose k items from n items with order and without repetition
math.pow()	Returns the value of x to the power of y
math.prod()	Returns the product of all the elements in an iterable
math.radians()	Converts a degree value into radians
math.remainder()	Returns the closest value that can make numerator completely divisible by the denominator
math.sin()	Returns the sine of a number
math.sinh()	Returns the hyperbolic sine of a number
math.sqrt()	Returns the square root of a number
math.tan()	Returns the tangent of a number
math.tanh()	Returns the hyperbolic tangent of a number
math.trunc()	Returns the truncated integer parts of a number

Math Constants

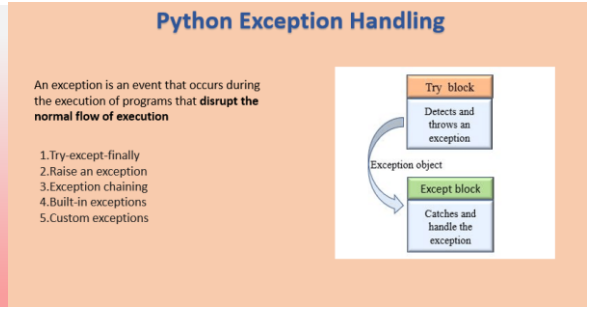
Constant	Description
math.e	Returns Euler's number (2.7182...)
math.inf	Returns a floating-point positive infinity
math.nan	Returns a floating-point NaN (Not a Number) value
math.pi	Returns PI (3.1415...)
math.tau	Returns tau (6.2831...)

- Python has a set of methods available for the file object





Method	Description
close()	Closes the file
detach()	Returns the separated raw stream from the buffer
fileno()	Returns a number that represents the stream, from the operating system's perspective
flush()	Flushes the internal buffer
isatty()	Returns whether the file stream is interactive or not
read()	Returns the file content
readable()	Returns whether the file stream can be read or not
readline()	Returns one line from the file
readlines()	Returns a list of lines from the file
seek()	Change the file position
seekable()	Returns whether the file allows us to change the file position
tell()	Returns the current file position
truncate()	Resizes the file to a specified size
writable()	Returns whether the file can be written to or not
write()	Writes the specified string to the file
writelines()	Writes a list of strings to the file



• **below shows built-in exceptions that are usually raised in Python:**

Exception	Description
ArithmeticError	Raised when an error occurs in numeric calculations
AssertionError	Raised when an assert statement fails
AttributeError	Raised when attribute reference or assignment fails
Exception	Base class for all exceptions
EOFError	Raised when the input() method hits an "end of file" condition (EOF)
FloatingPointError	Raised when a floating point calculation fails
GeneratorExit	Raised when a generator is closed (with the close() method)
ImportError	Raised when an imported module does not exist
IndentationError	Raised when indentation is not correct
IndexError	Raised when an index of a sequence does not exist
KeyError	Raised when a key does not exist in a dictionary
KeyboardInterrupt	Raised when the user presses Ctrl+c, Ctrl+z or Delete

LookupError	Raised when errors raised cant be found
MemoryError	Raised when a program runs out of memory
NameError	Raised when a variable does not exist
NotImplementedError	Raised when an abstract method requires an inherited class to override the method
OSError	Raised when a system related operation causes an error
OverflowError	Raised when the result of a numeric calculation is too large
ReferenceError	Raised when a weak reference object does not exist
RuntimeError	Raised when an error occurs that do not belong to any specific expectations
StopIteration	Raised when the next() method of an iterator has no further values
SyntaxError	Raised when a syntax error occurs
TabError	Raised when indentation consists of tabs or spaces
SystemError	Raised when a system error occurs
SystemExit	Raised when the sys.exit() function is called
TypeError	Raised when two different types are combined
UnboundLocalError	Raised when a local variable is referenced before assignment
UnicodeError	Raised when a Unicode problem occurs
UnicodeEncodeError	Raised when a Unicode encoding problem occurs
UnicodeDecodeError	Raised when a Unicode decoding problem occurs
UnicodeTranslateError	Raised when a Unicode translation problem occurs
ValueError	Raised when there is a wrong value in a specified data type
ZeroDivisionError	Raised when the second operator in a division is zero

- **Make a request to a web page, and print the response text:**

```
import requests

x = requests.get('https://w3schools.com/python/demopage.htm')

print(x.text)
```

Syntax

requests.methodname(params)

Method	Description
delete(url, args)	Sends a DELETE request to the specified url
get(url, params, args)	Sends a GET request to the specified url
head(url, args)	Sends a HEAD request to the specified url
patch(url, data, args)	Sends a PATCH request to the specified url
post(url, data, json, args)	Sends a POST request to the specified url
put(url, data, args)	Sends a PUT request to the specified url
request(method, url, args)	Sends a request of the specified method to the specified url