**NOMBRE DE LA ESCUELA**

INSTITUTO TECNOLÓGICO SUPERIOR DE CHICONTEPEC

**CARRERA**

INGENIERÍA EN SISTEMAS COMPUTACIONALES

**SEMESTRE**

CUARTO SEMESTRE

**MATERIA**

METODOS NUMERICOS

**TEMA**

REPORTE DE INSTALACION Y CONFIGURACION

**ALUMNA**

BLANCA FERNANDA DIEGO HERNANDEZ

**NOMBRE DEL DOCENTE**

ING. EFREN FLORES CRUZ

**LUGAR**

CHICONTEPEC

20/03/2020

MARCO TEORICO

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional.

Es un lenguaje interpretado, dinámico y multiplataforma.

Es administrado por la Python Software Fundación. Posee una licencia de código abierto, denominada Python Software Fundación License, que es compatible con la Licencia pública general de GNU a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriores.

Python fue creado a finales de los ochenta por Guido van Rossum en el Centro para las Matemáticas y la Informática (CWI, Centrum Wiskunde & Informatica), en los Países Bajos, como un sucesor del lenguaje de programación ABC, capaz de manejar excepciones e interactuar con el sistema operativo Amoeba.

El nombre del lenguaje proviene de la afición de su creador por los humoristas británicos Monty Python.

Van Rossum es el principal autor de Python, y su continuo rol central en decidir la dirección de Python es reconocido, refiriéndose a él como Benevolente Dictador Vitalicio (en inglés: Benevolent Dictator for Life, BDFL); sin embargo el 12 de julio de 2018 declinó de dicha situación de honor sin dejar un sucesor o sucesora y con una declaración altisonante.

Python es un lenguaje de programación multiparadigma. Esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: programación orientada a objetos, programación imperativa y programación funcional. Otros paradigmas están soportados mediante el uso de extensiones.

Python usa tipado dinámico y conteo de referencias para la administración de memoria.

Una característica importante de Python es la resolución dinámica de nombres; es decir, lo que enlaza un método y un nombre de variable durante la ejecución del programa (también llamado enlace dinámico de métodos).

Otro objetivo del diseño del lenguaje es la facilidad de extensión. Se pueden escribir nuevos módulos fácilmente en C o C++. Python puede incluirse en aplicaciones que necesitan una interfaz programable.

Aunque la programación en Python podría considerarse en algunas situaciones hostil a la programación funcional tradicional del Lisp, existen bastantes analogías

entre Python y los lenguajes minimalistas de la familia Lisp como puede ser Scheme.

Python fue diseñado para ser leído con facilidad. Una de sus características es el uso de palabras donde otros lenguajes utilizarían símbolos. Por ejemplo, los operadores lógicos `!`, `||` y `&&` en Python se escriben `not`, `or` y `and`, respectivamente. Curiosamente el lenguaje Pascal es junto con COBOL uno de los lenguajes con muy clara sintaxis y ambos son de la década del 70. La idea del código claro y legible no es algo nuevo.

El contenido de los bloques de código (bucles, funciones, clases, etc.) es delimitado mediante espacios o tabuladores, conocidos como indentación, antes de cada línea de órdenes pertenecientes al bloque.[25] Python se diferencia así de otros lenguajes de programación que mantienen como costumbre declarar los bloques mediante un conjunto de caracteres, normalmente entre llaves `{}`.[2627] Se pueden utilizar tanto espacios como tabuladores para indentar el código, pero se recomienda no mezclarlos.[28]

Función factorial en C (indentación opcional)

```
int factorial (int x)
{
    if (x < 0 || x % 1 != 0) {
        printf("x debe ser un numero entero mayor o igual a 0");
        return -1; //Error
    }
    if (x == 0) {
        return 1;
    }
    return x * factorial(x - 1);
}
```

Función factorial en Python (indentación obligatoria)

```
def factorial(x):
    assert x >= 0 and x % 1 == 0, "x debe ser un entero mayor o igual a 0."
    if x == 0:
        return 1
    else:
        return x * factorial(x - 1)
```

Debido al significado sintáctico de la indentación, cada instrucción debe estar contenida en una sola línea. No obstante, si por legibilidad se quiere dividir la instrucción en varias líneas, añadiendo una barra invertida `\` al final de una línea, se indica que la instrucción continúa en la siguiente.

Estas instrucciones son equivalentes:

```
lista=['valor 1','valor 2','valor 3']
cadena='Esto es una cadena bastante larga'
```

```
lista=['valor 1','valor 2' \
      ,'valor 3']
cadena='Esto es una cadena ' \
       'bastante larga'
```

Comentarios[editar]

Los comentarios se pueden poner de dos formas. La primera y más apropiada para comentarios largos es utilizando la notación ''' comentario ''', tres apóstrofos de apertura y tres de cierre. La segunda notación utiliza el símbolo `#`, y se extienden hasta el final de la línea.

El intérprete no tiene en cuenta los comentarios, lo cual es útil si deseamos poner información adicional en el código. Por ejemplo, una explicación sobre el comportamiento de una sección del programa.

```
'''
Comentario más largo en una línea en Python
'''
print ("Hola mundo") # También es posible añadir un comentario al final de una línea de código
```

Variables[editar]

Las variables se definen de forma dinámica, lo que significa que no se tiene que especificar cuál es su tipo de antemano y puede tomar distintos valores en otro momento, incluso de un tipo diferente al que tenía previamente. Se usa el símbolo `=` para asignar valores.

```
x = 1
x = "texto" # Esto es posible porque los tipos son asignados dinámicamente
```

Los nombres de variables pueden contener números y letras pero deben comenzar por una letra, además existen 28 palabras reservadas:[29]

| | | | |
|---|---|---|---|
| and | elif | global | or |
| assert | else | if | pass |
| break | except | import | print |
| class | exec | in | raise |
| continue | finally | is | return |
| def | for | lambda | try |
| del | from | not | while |

Los tipos de datos se pueden resumir en esta tabla:

| Tipo | Clase | Notas | Ejemplo |
|---|---|---|---|
| str | Cadena | Inmutable | 'Cadena' |
| unicode | Cadena | Versión Unicode de str | u'Cadena' |
| list | Secuencia | Mutable, puede contener objetos de diversos tipos | [4.0, 'Cadena', True] |
| tuple | Secuencia | Inmutable, puede contener objetos de diversos tipos | (4.0, 'Cadena', True) |
| set | Conjunto | Mutable, sin orden, no contiene duplicados | set([4.0, 'Cadena', True]) |
| frozenset | Conjunto | Inmutable, sin orden, no contiene duplicados | frozenset([4.0, 'Cadena', True]) |
| dict | Mapping | Grupo de pares clave:valor | {'key1': 1.0, 'key2': False} |

| | | | |
|---|---|---|---|
| int | Número entero | Precisión fija, convertido en long en caso de overflow. | 42 |
| long | Número entero | Precisión arbitraria | 42L ó 456966786151987643L |
| float | Número decimal | Coma flotante de doble precisión | 3.1415927 |
| complex | Número complejo | Parte real y parte imaginaria j. | (4.5 + 3j) |
| bool | Booleano | Valor booleano verdadero o falso | True o False |

Mutable: si su contenido (o dicho valor) puede cambiarse en tiempo de ejecución.

Inmutable: si su contenido (o dicho valor) no puede cambiarse en tiempo de ejecución.

Condicionales

Una sentencia condicional (if) ejecuta su bloque de código interno solo si se cumple cierta condición. Se define usando la palabra clave `if` seguida de la condición, y el bloque de código. Condiciones adicionales, si las hay, se introducen usando `elif` seguida de la condición y su bloque de código. Todas las condiciones se evalúan secuencialmente hasta encontrar la primera que sea verdadera, y su bloque de código asociado es el único que se ejecuta. Opcionalmente, puede haber un bloque final (la palabra clave `else` seguida de un bloque de código) que se ejecuta solo cuando todas las condiciones fueron falsas.

```
>>> verdadero = True
>>> if verdadero: # No es necesario poner "verdadero == True"
...     print "Verdadero"
... else:
...     print "Falso"
...
Verdadero
>>> lenguaje = "Python"
```

>>> if lenguaje == "C": # lenguaje no es "C", por lo que este bloque se obviará y evaluará la siguiente condición

... print "Lenguaje de programación: C"

... elif lenguaje == "Python": # Se pueden añadir tantos bloques "elif" como se quiera

... print "Lenguaje de programación: Python"

... else: # En caso de que ninguna de las anteriores condiciones fuera cierta, se ejecutaría este bloque

... print "Lenguaje de programación: indefinido"

...

Lenguaje de programación: Python

>>> if verdadero and lenguaje == "Python": # Uso de "and" para comprobar que ambas condiciones son verdaderas

... print "Verdadero y Lenguaje de programación: Python"

...

Verdadero y Lenguaje de programación: Python

Bucle for

El bucle for es similar a foreach en otros lenguajes. Recorre un objeto iterable, como una lista, una tupla o un generador, y por cada elemento del iterable ejecuta el bloque de código interno. Se define con la palabra clave `for` seguida de un nombre de variable, seguido de `in`, seguido del iterable, y finalmente el bloque de código interno. En cada iteración, el elemento siguiente del iterable se asigna al nombre de variable especificado:

>>> lista = ["a", "b", "c"]

>>> for i in lista: # Iteramos sobre una lista, que es iterable

... print i

...

a

b

c

>>> cadena = "abcdef"

>>> for i in cadena: # Iteramos sobre una cadena, que también es iterable

... print(i, end=', ') # Añadiendo end=', ' al final hacemos que no introduzca un salto de línea, sino un espacio

...

a b c d e f

Bucle while

El bucle while evalúa una condición y, si es verdadera, ejecuta el bloque de código interno. Continúa evaluando y ejecutando mientras la condición sea verdadera. Se define con la palabra clave `while` seguida de la condición, y a continuación el bloque de código interno:

>>> numero = 0

>>> while numero < 10:

... print (numero),

... numero += 1 # Un buen programador modificará las variables de control al finalizar el ciclo while

...

0 1 2 3 4 5 6 7 8 9

Listas y Tuplas

Para declarar una lista se usan los corchetes `[]`, en cambio, para declarar una tupla se usan los paréntesis `()`. En ambas los elementos se separan por comas, y en el caso de las tuplas es necesario que tengan como mínimo una coma.

Tanto las listas como las tuplas pueden contener elementos de diferentes tipos. No obstante las listas suelen usarse para elementos del mismo tipo en cantidad variable mientras que las tuplas se reservan para elementos distintos en cantidad fija.

Para acceder a los elementos de una lista o tupla se utiliza un índice entero (empezando por "0", no por "1"). Se pueden utilizar índices negativos para acceder elementos a partir del final.

Las listas se caracterizan por ser mutables, es decir, se puede cambiar su contenido en tiempo de ejecución, mientras que las tuplas son inmutables ya que no es posible modificar el contenido una vez creada.

Listas

```
>>> lista = ["abc", 42, 3.1415]
>>> lista[0] # Acceder a un elemento por su índice
'abc'
>>> lista[-1] # Acceder a un elemento usando un índice negativo
3.1415
>>> lista.append(True) # Añadir un elemento al final de la lista
>>> lista
['abc', 42, 3.1415, True]
>>> del lista[3] # Borra un elemento de la lista usando un índice (en este caso: True)
>>> lista[0] = "xyz" # Re-asignar el valor del primer elemento de la lista
>>> lista[0:2] # Mostrar los elementos de la lista del índice "0" al "2" (sin incluir este último)
['xyz', 42]
>>> lista_anidada = [lista, [True, 42L]] # Es posible anidar listas
>>> lista_anidada
[['xyz', 42, 3.1415], [True, 42L]]
>>> lista_anidada[1][0] # Acceder a un elemento de una lista dentro de otra lista (del segundo elemento, mostrar el primer elemento)
True
```

Tuplas

```
>>> tupla = ("abc", 42, 3.1415)
>>> tupla[0] # Acceder a un elemento por su índice
'abc'
>>> del tupla[0] # No es posible borrar (ni añadir) un elemento en una tupla, lo que provocará una excepción
( Excepción )
>>> tupla[0] = "xyz" # Tampoco es posible re-asignar el valor de un elemento en una tupla, lo que también provocará una excepción
( Excepción )
```

>>> tupla[0:2] # Mostrar los elementos de la tupla del índice "0" al "2" (sin incluir este último)

('abc', 42)

>>> tupla_anidada = (tupla, (True, 3.1415)) # También es posible anidar tuplas

>>> 1, 2, 3, "abc" # Esto también es una tupla, aunque es recomendable ponerla entre paréntesis (recuerda que requiere, al menos, una coma)

(1, 2, 3, 'abc')

>>> (1) # Aunque entre paréntesis, esto no es una tupla, ya que no posee al menos una coma, por lo que únicamente aparecerá el valor

1

>>> (1,) # En cambio, en este otro caso, sí es una tupla

(1,)

>>> (1, 2) # Con más de un elemento no es necesaria la coma final

(1, 2)

>>> (1, 2,) # Aunque agregarla no modifica el resultado

(1, 2)

Sentencia Switch Case

Si bien Python no tiene la estructura Switch, hay varias formas de realizar la operación típica que realizaríamos con una sentencia switch case.

Usando if, elif, else[editar]

Podemos usar la estructura de la siguiente manera:

```
>>> if condicion1:
...     hacer1
>>> elif condicion2:
...     hacer2
>>> elif condicion3:
...     hacer3
>>> else:
...     hacer
```

En esa estructura se ejecutara controlando la condicion1, si no se cumple pasara a la siguiente y así sucesivamente hasta entrar en el else. Un ejemplo práctico seria:

```
>>> def calculo(op,a,b):
...     if 'sum' == op:
...         return a + b
...     elif 'rest' == op:
...         return a - b
...     elif 'mult' == op:
...         return a * b
...     elif 'div' == op:
...         return a/b
...     else:
...         return None
>>>
>>> print(calculo('sum',3,4))
```

Podriamos decir que el lado negativo de la sentencia armada con if, elif y else es que si la lista de posibles operaciones es muy larga, las tiene que recorrer una por una hasta llegar a la correcta.

Usando diccionario[editar]

Podemos usar un diccionario para el mismo ejemplo:

```
>>> def calculo(op,a,b):
...     return {
...         'sum': lambda: a + b,
...         'rest': lambda: a - b,
...         'mult': lambda: a * b,
...         'div': lambda: a/b
...     }.get(op, lambda: None)()
>>>
>>> print(calculo('sum',3,4))
```

De esta manera, si las opciones fueran muchas, no recorrería todas; solo iría directamente a la operación buscada en la última línea .get(op, lambda: None)() estamos dando la opción por defecto.

Conjuntos

Los conjuntos se construyen mediante `set(items)` donde items es cualquier objeto iterable, como listas o tuplas. Los conjuntos no mantienen el orden ni contienen elementos duplicados.

Se suelen utilizar para eliminar duplicados de una secuencia, o para operaciones matemáticas como intersección, unión, diferencia y diferencia simétrica.

```
>>> conjunto_inmutable = frozenset(["a", "b", "a"]) # Se utiliza una lista como objeto iterable

>>> conjunto_inmutable

frozenset(['a', 'b'])

>>> conjunto1 = set(["a", "b", "a"]) # Primer conjunto mutable

>>> conjunto1

set(['a', 'b'])

>>> conjunto2 = set(["a", "b", "c", "d"]) # Segundo conjunto mutable

>>> conjunto2

set(['a', 'c', 'b', 'd']) # Recuerda, no mantienen el orden, como los diccionarios

>>> conjunto1 & conjunto2 # Intersección

set(['a', 'b'])

>>> conjunto1 | conjunto2 # Unión

set(['a', 'c', 'b', 'd'])

>>> conjunto1 - conjunto2 # Diferencia (1)

set([])

>>> conjunto2 - conjunto1 # Diferencia (2)

set(['c', 'd'])

>>> conjunto1 ^ conjunto2 # Diferencia simétrica

set(['c', 'd'])
```

Listas por comprensión

Una lista por comprensión (en inglés: list comprehension) es una expresión compacta para definir listas. Al igual que `lambda`, aparece en lenguajes funcionales. Ejemplos:

```
>>> range(5) # La función "range" devuelve una lista, empezando en 0 y terminando con el número indicado menos uno
[0, 1, 2, 3, 4]
>>> [i*i for i in range(5)] # Por cada elemento del rango, lo multiplica por sí mismo y lo agrega al resultado
[0, 1, 4, 9, 16]
>>> lista = [(i, i + 2) for i in range(5)]
>>> lista
[(0, 2), (1, 3), (2, 4), (3, 5), (4, 6)]
```

Funciones

Las funciones se definen con la palabra clave `def`, seguida del nombre de la función y sus parámetros. Otra forma de escribir funciones, aunque menos utilizada, es con la palabra clave `lambda` (que aparece en lenguajes funcionales como Lisp).

El valor devuelto en las funciones con `def` será el dado con la instrucción `return`.

`def`:

```
>>> def suma(x, y = 2):
...     return x + y # Retornar la suma del valor de la variable "x" y el valor de "y"
...
>>> suma(4) # La variable "y" no se modifica, siendo su valor: 2
6
>>> suma(4, 10) # La variable "y" sí se modifica, siendo su nuevo valor: 10
14
```

`lambda`:

```
>>> suma = lambda x, y = 2: x + y
```

>>> suma(4) # La variable "y" no se modifica, siendo su valor: 2

6

>>> suma(4, 10) # La variable "y" sí se modifica, siendo su nuevo valor: 10

14

Clases

Las clases se definen con la palabra clave `class`, seguida del nombre de la clase y, si hereda de otra clase, el nombre de esta.

En Python 2.x era recomendable que una clase heredase de "Object", en Python 3.x ya no hace falta.

En una clase un "método" equivale a una "función", y un "atributo" equivale a una "variable".

"__init__" es un método especial que se ejecuta al instanciar la clase, se usa generalmente para inicializar atributos y ejecutar métodos necesarios. Al igual que todos los métodos en Python, debe tener al menos un parámetro, generalmente se utiliza `self`. El resto de parámetros serán los que se indiquen al instanciar la clase.

Los atributos que se desee que sean accesibles desde fuera de la clase se deben declarar usando `self.` delante del nombre.

En Python no existe el concepto de encapsulación,[30] por lo que el programador debe ser responsable de asignar los valores a los atributos

```
>>> class Persona():
...     def __init__(self, nombre, edad):
...         self.nombre = nombre # Un atributo cualquiera
...         self.edad = edad # Otro atributo cualquiera
...     def mostrar_edad(self): # Es necesario que, al menos, tenga un parámetro, generalmente: "self"
...         print(self.edad) # mostrando un atributo
...     def modificar_edad(self, edad): # Modificando Edad
...         if edad < 0 or edad > 150: # Se comprueba que la edad no sea menor de 0 (algo imposible), ni mayor de 150 (algo realmente difícil)
...             return False
...         else: # Si está en el rango 0-150, entonces se modifica la variable
```

```
...         self.edad = edad # Se modifica la edad
...
>>> p = Persona('Alicia', 20) # Instanciar la clase, como se puede ver, no se especifica el valor de "self"
>>> p.nombre # La variable "nombre" del objeto sí es accesible desde fuera
'Alicia'
>>> p.nombre = 'Andrea' # Y por tanto, se puede cambiar su contenido
>>> p.nombre
'Andrea'
>>> p.mostrar_edad() # Se llama a un método de la clase
20
>>> p.modificar_edad(21) # Es posible cambiar la edad usando el método específico que hemos hecho para hacerlo de forma controlada
>>> p.mostrar_edad()
21
```

Módulos

Existen muchas propiedades que se pueden agregar al lenguaje importando módulos, que son "minicódigos" (la mayoría escritos también en Python) que proveen de ciertas funciones y clases para realizar determinadas tareas. Un ejemplo es el módulo Tkinter[31], que permite crear interfaces gráficas basadas en la biblioteca Tk. Otro ejemplo es el módulo os, que provee acceso a muchas funciones del sistema operativo. Los módulos se agregan a los códigos escribiendo `import` seguida del nombre del módulo que queramos usar.[32]

Interfaz al sistema operativo[editar]

El módulo os provee funciones para interactuar con el sistema operativo:

```
>>> import os # Módulo que provee funciones del sistema operativo
>>> os.name # Devuelve el nombre del sistema operativo
'posix'
>>> os.mkdir("/tmp/ejemplo") # Crea un directorio en la ruta especificada
>>> import time # Módulo para trabajar con fechas y horas
```

>>> time.strftime("%Y-%m-%d %H:%M:%S") # Dándole un cierto formato, devuelve la fecha y/u hora actual

'2010-08-10 18:01:17'

Para tareas de administración de archivos, el módulo shutil provee una interfaz de más alto nivel:

>>> import shutil

>>> shutil.copyfile('datos.db', 'informacion.db')

'informacion.db'

>>> shutil.move('/build/programas', 'dir_progs')

'dir_progs'

Comodines de archivos[editar]

El módulo glob provee una función para crear listas de archivos a partir de búsquedas con comodines en carpetas:

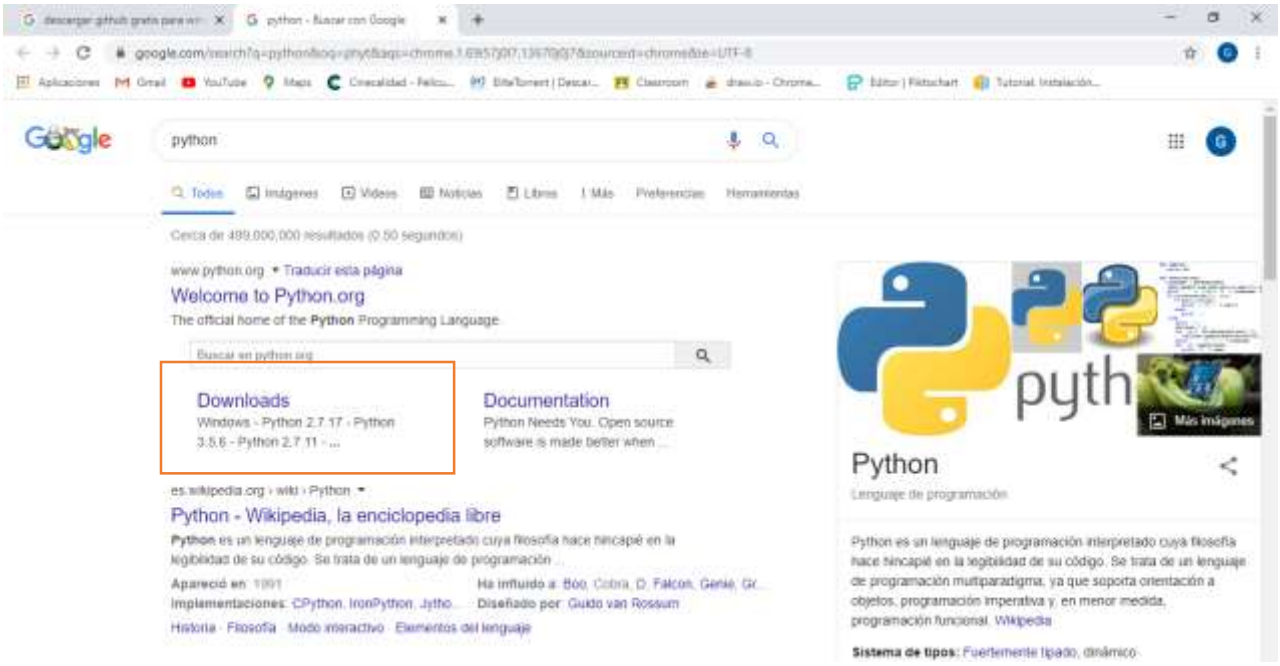
>>> import glob

>>> glob.glob('*.py')

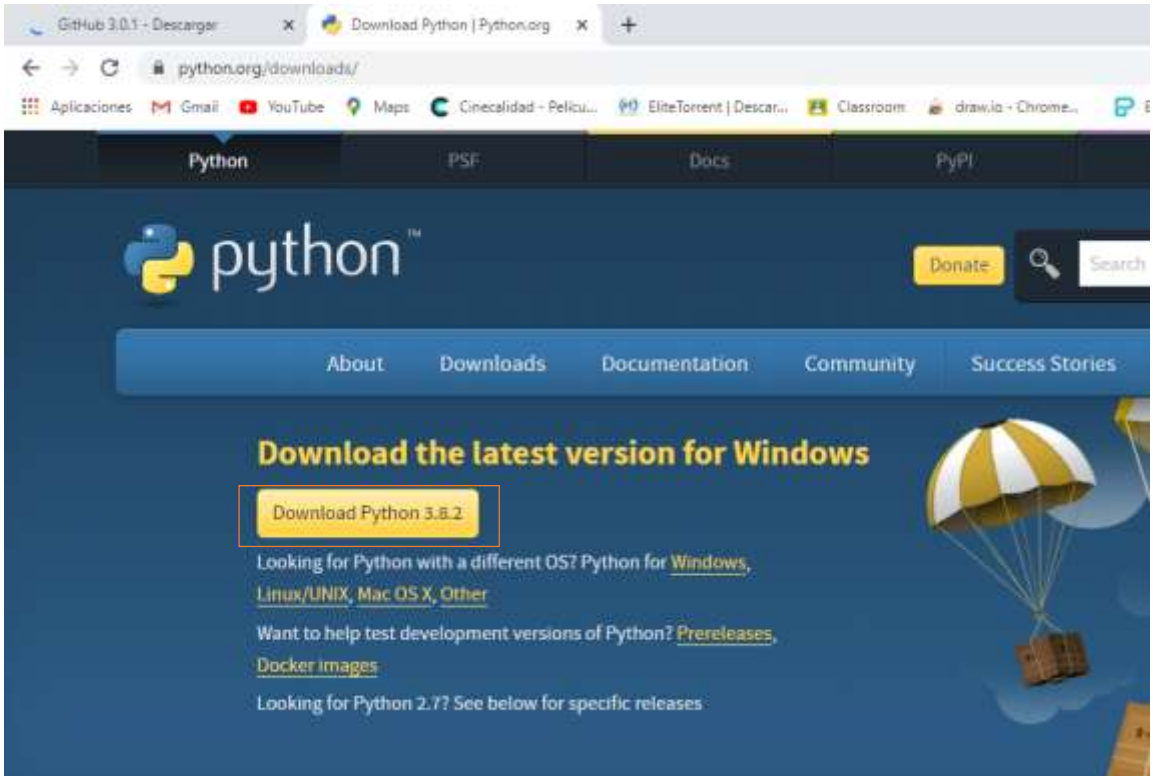
['numeros.py', 'ejemplo.py', 'ejemplo2.py']

METODOLOGIA

1.-Primero ingresamos al navegador y dentro del buscador ingresamos “descargar Python” nos aparecerá la página que aquí damos clic en la parte seleccionada y nos abrirá otra página.

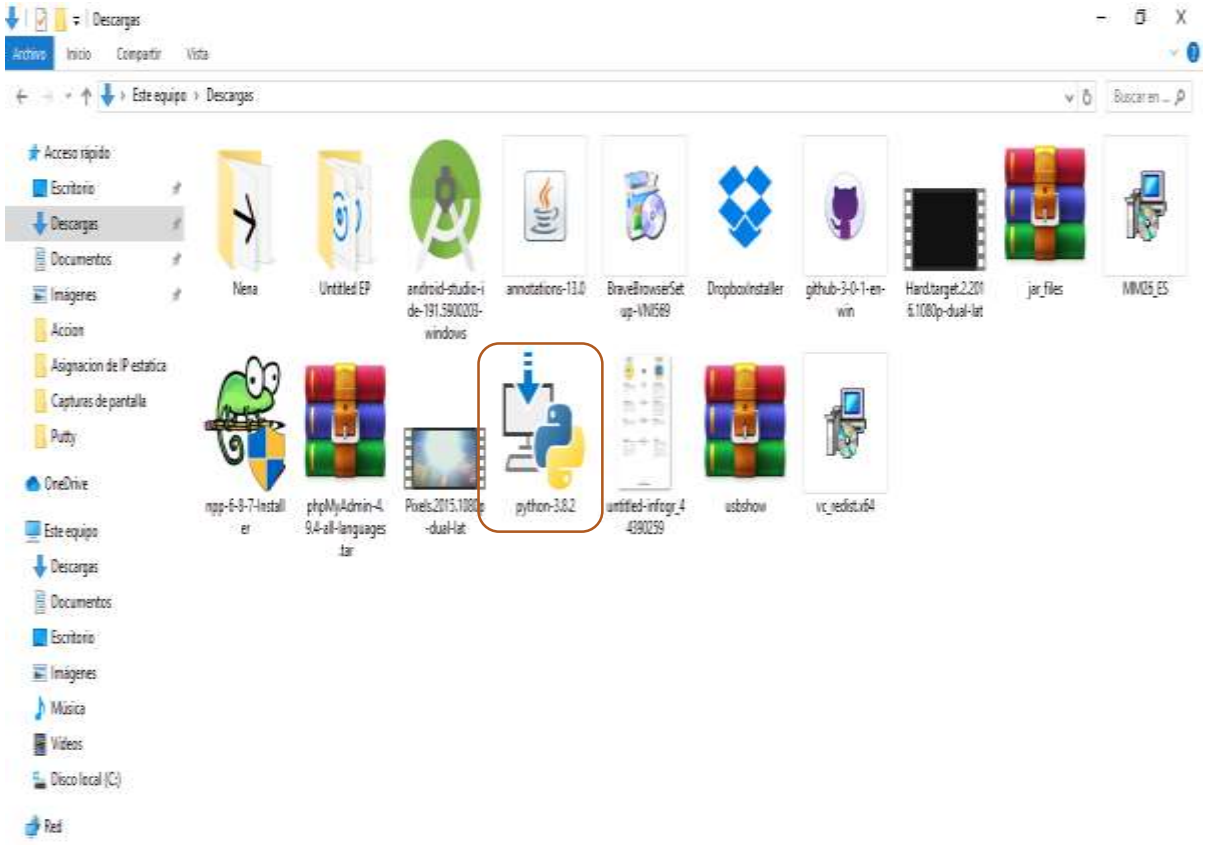


2.-En esta página seleccionamos donde se está marcado y se comienza la descarga



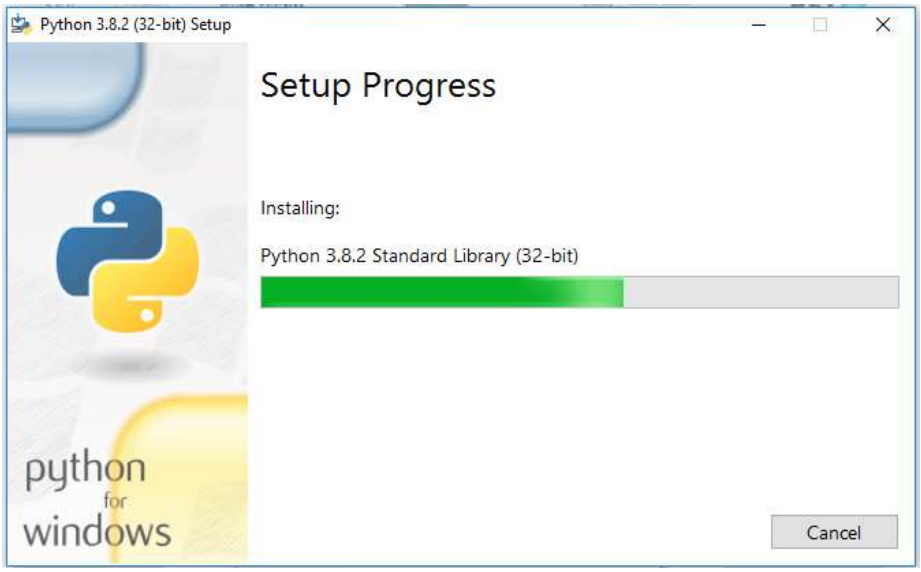
3.-En esta parte se muestra el proceso durante la descarga

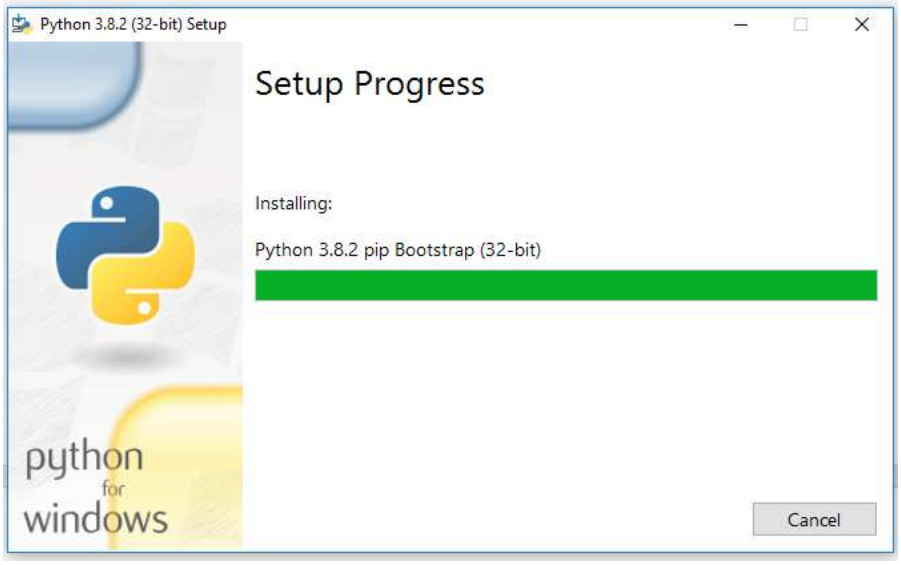
4.-Una vez descargado se procese a realizar la instalación para esto se va a donde se guarda el programa y esta muestra el programa descargado



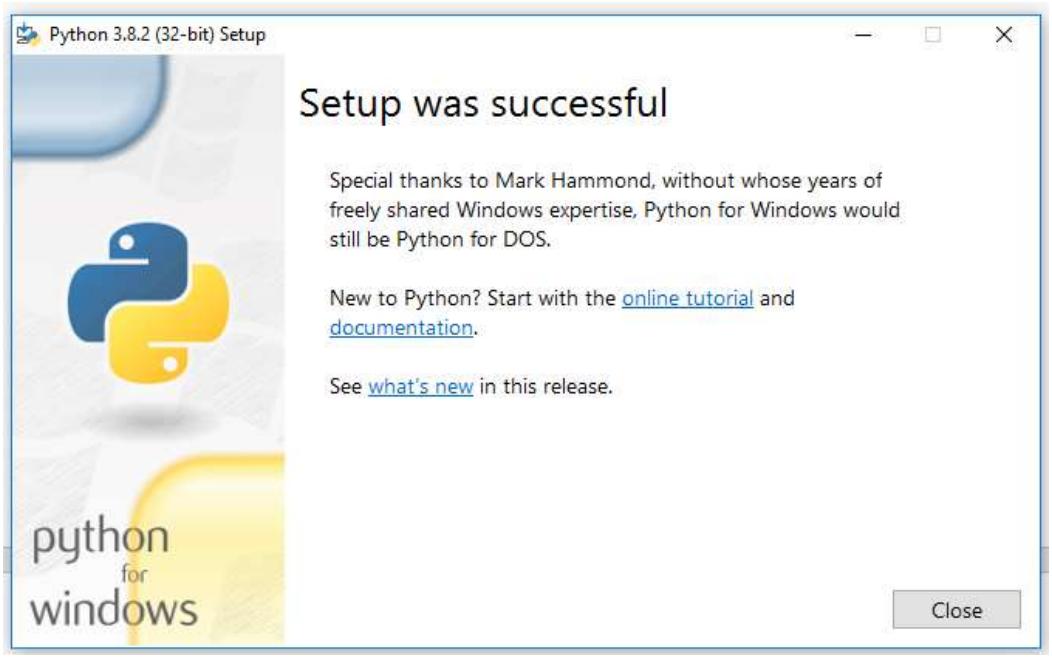
5.-Abrimos el programa y comenzamos la instalación

6.- Una vez seleccionada la opción anterior se comienza a realizar la instalación de nuestro programa aquí se va mostrando el proceso que lleva la instalación



7.- Se muestra la finalización de la instalación de nuestro programa



BIBLIOGRAFIA

https://www.python.org/downloads/

https://es.wikipedia.org/wiki/Python