**--------verificar lo intalado---**

pip -V

pip install --upgrade pip

which virtualenv

pip freeze: Lista las extensiones instaladas en nuestro systema o

entorno virtual activo

pip install -r requirements.txt (se crea el .txt antes y se deja lo que se guado para instalar )

--------ENTORNO VIRTUAL----

luego seguí estos pasos.

1. pip Install virtualenv
2. Pip Install virtualenvwrapper-win:
3. una vez echo esto anote el comando que dice el profe con una ligera modificación  
   estando en la carpeta “platzi-ventas” ejecuto el comando:

* **virtualenv --python=python venv**

1. si ya ha respondido la consola con **done**, ingreso a la carpeta con el nombre que se creo **“venv”**

* **cd venv/Scripts**

1. ejecuto el archivo activate  
   **activate**, inmediatamente inicializa el ambiente virtual tal como al profe!!

… \CURSO PLATZI\CURSO\_PYTHON\_3\PLATZI-VENTAS\venv\Script **activate**

Y por fin ingresas al entorno virtual

**(venv)** C:\Users\..\CURSO PLATZI\CURSO\_PYTHON\_3\PLATZI-VENTAS\venv

* luego regresa a la carpeta donde esta es setup.py en la Carpeta PLATZI-VENTAS y ejecuta :

**pip install --editable .**

**Funciones principales en Turtle Graphics**

<https://www.programoergosum.com/cursos-online/raspberry-pi/245-turtle-graphics-en-python-con-raspberry-pi/mi-primera-figura>

Las funciones principales para animar nuestro objeto son las siguientes:

**forward(distance)**: Avanzar una determinada cantidad de píxeles.  
**backward(distance)**: Retroceder una determinada cantidad de píxeles.  
**left(angle)**: Girar hacia la izquierda un determinado ángulo.  
**right(angle)**: Girar hacia la derecha un determinado ángulo.

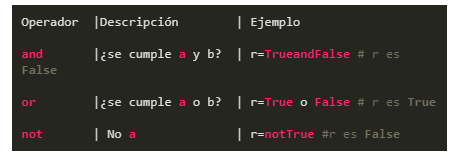
Por otro lado, puede que en ocasiones queramos desplazarnos de un punto a otro sin dejar rastro. Para ello utilizaremos las siguientes funciones:

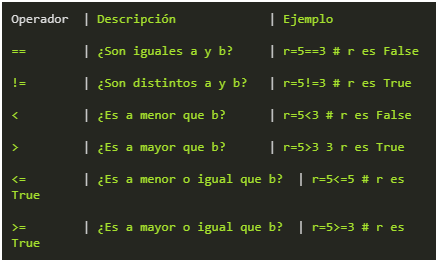
**home(distance)**: Desplazarse al origen de coordenadas.  
**goto((x, y))**: Desplazarse a una coordenada en concreto.  
**pendown()**: Subir el lápiz para no mostrar el rastro.  
**penup()**: Bajar el lápiz para mostrar el rastro.

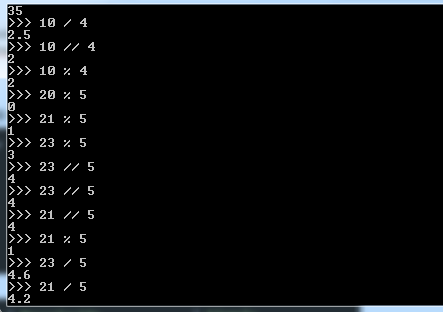
Por último, puede que queramos cambiar el color o tamaño del lápiz. En ese caso utilizaremos las siguientes funciones gráficas:

**shape(‘turtle’)**: Cambia al objeto tortuga.  
**pencolor(color)**: Cambiar al color especificado.  
**pensize(dimension)**: Tamaño de la punta del lápiz.

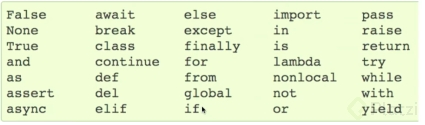
* **OPERADOS MATEMATICOS**







* **PALABRAS RESERVADAS**



* **FUNCIONES IMPORTANTES**

<https://docs.python.org/3/library/functions.html#any>

**Operaciones con Strings en Python**

Los strings tienen varios métodos que nosotros podemos utilizar.

* **upper**: convierte todo el **string** a mayúsculas
* **lower:** convierte todo el **string** a minúsculas
* **find:** encuentra el indice en donde existe un patrón que nosotros definimos
* **startswith:** significa que empieza con algún patrón.
* **endswith:** significa que termina con algún patrón
* **capitalize:** coloca la primera letra en mayúscula y el resto en minúscula
* **in y not in** nos permite saber con cualquier secuencia sin una

subsecuencia o substrings se encuentra adentro de la secuencia mayor.

* **dir:** Nos dice todos los métodos que podemos utilizar dentro de un objeto.
* **help:** nos imprime en pantalla el *docstrings* o comentario de ayuda o instrucciones que posee la función. Casi todas las funciones en Python las tienen.

Ej:



Para limpiar en el cmd de windows usa:

>>> import os

>>> clear = lambda: os.system('cls')

>>> clear()

y despues ya puedes usar solo

>>> clear()

* **METODOS STRING**

|  |  |
| --- | --- |
| * help(str) dir(str) help(str.splitlines) import time help(time)   capitalize() casefold() center(width[, fillchar]) count(sub[, start[, end]]) encode(encoding=“utf-8”, errors=“strict”) endswith(suffix[, start[, end]]) expandtabs(tabsize=8) find(sub[, start[, end]]) format(\*args, \*\*kwargs) format\_map(mapping) index(sub[, start[, end]]) isalnum() isalpha() isdecimal() isdigit() isidentifier() islower() translate(table) upper() zfill(width) | isnumeric() isprintable() isspace() istitle() isupper() join(iterable) ljust(width[, fillchar]) lower() lstrip([chars]) maketrans(x[, y[, z]]) partition(sep) replace(old, new[, count]) rfind(sub[, start[, end]]) rindex(sub[, start[, end]]) rjust(width[, fillchar]) rpartition(sep) rsplit(sep=None, maxsplit=-1) rstrip([chars]) split(sep=None, maxsplit=-1) splitlines([keepends]) startswith(prefix[, start[, end]]) strip([chars]) swapcase() title() |

**CRUD**

(C)reate

(R)ead or retreive

(U)pdate

(D)elete

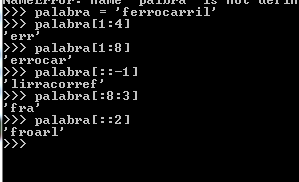
**Operaciones con strings: Slices en python**

Los *slices* en Python nos permiten manejar secuencia de una manera poderosa.

**Slices** en español significa ““rebanada””, si tenemos una secuencia de elementos y queremos una rebanada tenemos una sintaxis para definir que pedazos queremos de esa secuencia.

secuencia[comienzo:final:pasos]

EJ:



**For loops**

Las iteraciones es uno de los conceptos **más importantes** en la programación. En **Python** existen muchas manera de iterar pero las dos principales son los **for loops** y **while loops**.

Los **for loops** nos permiten iterar a través de una secuencia y los while loops nos permiten iterara hasta cuando una condición se vuelva falsa.

**for loops**:

* Tienen dos keywords break y continue que nos permiten salir anticipadamente de la iteración
* Se usan cuando se quiere ejecutar varias veces una o varias instrucciones.
* for [variable] in [secuencia]:

Es una convención usar la letra i como variable en nuestro for, pero podemos colocar la que queramos.

range: Nos da un objeto rango, es un iterador sobre el cual podemos generar secuencias

**While loops**

Al igual que las for loops, las **while loops** nos sirve para iterar, pero las for loops nos sirve para iterar a lo largo de una secuencia mientras que las **while loops** nos sirve para iterar mientras una condición sea verdadera.

Si no tenemos un mecanismo para convertir el mecanismo en falsedad, entonces nuestro while loops se ira al infinito(infinite loop).

**Uso de listas**

Python y todos los lenguajes nos ofrecen *constructos* mucho más poderoso, haciendo que el desarrollo de nuestro software sea

* Más sofisticado
* Más legible
* Más fácil de implementar

Estos *constructos* se llaman **Estructuras de Datos** que nos permiten agrupar de distintas manera varios valores y elementos para poderlos manipular con mayor facilidad.

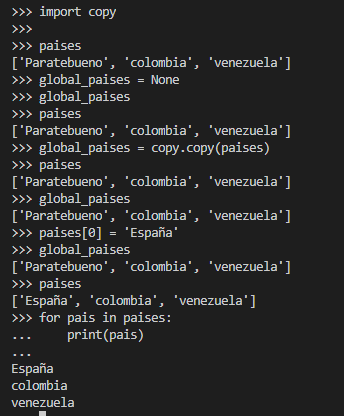
Las **listas** las vas a utilizar durante toda tu carrera dentro de la programación e ingeniería de Software.

Las **listas** son una secuencia de valores. A diferencia de los *strings*, las **listas** pueden tener cualquier tipo de valor. También, a diferencia de los strings, son mutables, podemos agregar y eliminar elementos.

En Python, las listas son referenciales. Una lista no guarda en memoria los objetos, sólo guarda la **referencia** hacia donde viven los objetos en memoria

Se inician con [] o con la *built-in function* list.

**Uso de import copy:**



**Operaciones con listas**

Ahora que ya entiendes cómo funcionan las **listas**, podemos ver qué tipo de operaciones y métodos podemos utilizar para modificarlas, manipularlas y realizar diferentes tipos de cómputos con esta Estructura de Datos.

* El operador **+(suma)** concatena dos o más listas.
* El operador \***(multiplicación)** repite los elementos de la misma lista tantas veces los queramos multiplicar

Sólo podemos utilizar **+(suma)** y \***(multiplicación)**.

Las listas tienen varios métodos que podemos utilizar.

* **append** nos permite añadir elementos a listas. Cambia el tamaño de la lista.
* **pop** nos permite sacar el último elemento de la lista. También recibe un índice y esto nos permite elegir qué elemento queremos eliminar.
* **sort** modifica la propia lista y ordenarla de mayor a menor. Existe otro método llamado **sorted,** que también ordena la lista, pero genera una nueva instancia de la lista
* **del** nos permite eliminar elementos vía indices, funciona con *slices*
* **remove** nos permite es pasarle un valor para que Python compare internamente los valores y determina cuál de ellos hace match o son iguales para eliminarlos.

Ej:

**>>> a = list(range(0, 100, 2))**

**>>> a**

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]

**>>> b= list(range(0, 10, 2))**

**>>> a + b**

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 0, 2, 4, 6, 8]

**>>> b \* 2**

[0, 2, 4, 6, 8, 0, 2, 4, 6, 8]

**>>> a/b**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unsupported operand type(s) for /: 'list' and 'list'

**>>> a % b**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

**>>> a – b**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

**>>> fruits = list()**

**>>> fruits**

[]

>>> **fruits.append('apple')**

**>>> fruits**

['apple']

**>>> fruits.append('banana')**

**>>> fruits**

['apple', 'banana']

**>>> len(fruits)**

2

**>>> fruits.append('kiwi')**

**>>> len(fruits)**

3

**>>> fruits**

['apple', 'banana', 'kiwi']

**>>> some\_fruit = fruits.pop()**

**>>> some\_fruit**

'kiwi'

**>>> fruits**

['apple', 'banana']

**>>> some\_fruit = fruits.pop(0)**

**>>> some\_fruit**

'apple'

**>>> fruits**

['banana']

**>>> del fruits[0]**

**>>> fruits**

[] borro el ultimo de la lista

>>>

**import random**

**>>>**

**>>> random\_numbers = []**

**>>> for i in range(10): random\_numbers.append(random.randint(0, 15))**

**>>> random\_numbers**

[4, 8, 7, 9, 0, 2, 15, 8, 3, 12]

**>>> ordenar = sorted(random\_numbers)**

**>>> random\_numbers**

[4, 8, 7, 9, 0, 2, 15, 8, 3, 12]

**>>> ordenar**

[0, 2, 3, 4, 7, 8, 8, 9, 12, 15]

**>>> random\_numbers.sort()**

**>>> random\_numbers**

[0, 2, 3, 4, 7, 8, 8, 9, 12, 15]

***VER TODOS LOS METODOS DE LA LIST***

**>>> dir(random\_numbers)**

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

list.clear()

>>> my\_list = list(range(20))

my\_list.insert(2, 30)

print("new list:", my\_list)

newlist: [0, 1, 30, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

**Diccionarios**

Los diccionarios se conocen con diferentes nombres a lo largo de los lenguajes de programación como HashMaps, Mapas, Objetos, etc. En Python se conocen como **Diccionarios**.

Un diccionario es similar a una lista sabiendo que podemos acceder a través de un indice, pero en el caso de las listas este índice debe ser un número entero. Con los diccionarios puede ser cualquier objeto, normalmente los verán con **strings** para ser más explicitos, pero funcionan con muchos tipos de llaves…

Un diccionario es una asociación entre llaves(**keys**) y valores(**values**) y la referencia en Python es muy precisa. Si abres un diccionario verás muchas palabras y cada palabra tiene su definición.

Para iniciar un diccionario se usa {} o con la función dict

Estos también tienen varios métodos. Siempre puedes usar la función dir para saber todos los métodos que puedes usar con un objeto.

Si queremos ciclar a lo largo de un diccionario tenemos las opciones:

**keys**: nos imprime una lista de las llaves  
**values** nos imprime una lista de los valores  
**items**. nos manda una lista de tuplas de los valores

EJ :

rae = {}

>>> rae['pizza'] = 'la comida mas rica del mundo'

>>> rae

{'pizza': 'la comida mas rica del mundo'}

>>> rae['pasta'] = 'la comida mas sabrosa de italia'

>>> rae

{'pizza': 'la comida mas rica del mundo', 'pasta': 'la comida mas sabrosa de italia'}

>>> rae['pizza']

'la comida mas rica del mundo'

>>> rae['pasta']

'la comida mas sabrosa de italia'

***ERROR:***

>>>>>> a = rae.get('helado', None)

>>> print(a) = None

>>> a = rae.get('pizza', None)

>>> print(a) = la comida mas rica del mundo

>>> rae.keys() = dict\_keys(['pizza', 'pasta'])

>>> rae.values()

dict\_values(['la comida mas rica del mundo', 'la comida mas sabrosa de italia'])

>>> rae.items()

dict\_items([('pizza', 'la comida mas rica del mundo'), ('pasta', 'la comida mas sabrosa de italia')]) se hace una tupla

>>> for key in rae.keys():

... print(key)

pizza

pasta

>>> for key in rae.values():

... print(key)

la comida mas rica del mundo

la comida mas sabrosa de Italia

**>>> for key, value in rae.items():**

... print(key, value)...

pizza la comida mas rica del mundo

pasta la comida mas sabrosa de Italia

**todas las funcions:**

>>> dir(rae)

['\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']

**Tuplas y conjuntos en código**

En esta clase practicaremos en código lo aprendido en la clase anterior sobre tuplas(**tuples**) y conjuntos(**sets**) para que sea mucho más claro entenderlo.

Ej:

>>> b = (1, 2, 3)

>>> type(b)

<class 'tuple'>

>>> a[0]

>>> a = (1, 1, 1, 2, 4, 2, 3)

>>> a.count(1) = 3

>>> a.count(2) = 2

>>> a.index(2) = 3

>>> a.index(3) = 6

Operadores para **sets** en Python:

>>> a = set([1, 2, 3])

>>> b = {3, 4, 5}

>>> type(a)

<class 'set'>

>>> type(b)

<class 'set'>

>>> a.add(3)

>>> a = {1, 2, 3} no admite repetidos

>>> a.add(20)

>>> a = {1, 2, 3, 20}

>>> b = {3, 4, 5}

>>> a.union(b) = {1, 2, 3, 20, 4, 5}

>>> a.difference(b)= {1, 2, 20}

>>> b.difference(a)= {4, 5}

>>> a.remove(20)

>>> a = {1, 2, 3}

>>> A = {1, 2, 3} # conjunto A

>>> B = {3, 4 ,5} # conjunto B

>>> A | B #unión

{1, 2, 3, 4, 5}

>>> A & B # intersección

{3}

>>> A - B # diferencia entre A y B

{1, 2}

>>> B - A # diferencia entre B y A

{4, 5}

**Introducción al módulo collections**

El módulo collections nos brinda un conjunto de objetos primitivos que nos permiten extender el comportamiento de las built-in collections que poseé Python y nos otorga estructuras de datos adicionales. Por ejemplo, si queremos extender el comportamiento de un diccionario, podemos extender la clase UserDict; para el caso de una lista, extendemos UserList; y para el caso de strings, utilizamos UserString.

Por ejemplo, si queremos tener el comportamiento de un diccionario podemos escribir el siguiente código:

**classSecretDict**(collections.UserDict):**def\_password\_is\_valid**(**self**, password):

…

**def\_get\_item**(**self**, key):

…

**def\_\_getitem\_\_**(**self**, key):

password, key = key.split(‘:’)

**if** **self**.\_password\_is\_valid(password):

**return** **self**.\_get\_item(key)

**returnNone**

my\_secret\_dict = SecretDict(...)

my\_secret\_dict[‘some\_password:some\_key’] # si el password es válido, regresa el valor

Otra estructura de datos que vale la pena analizar, es namedtuple. Hasta ahora, has utilizado tuples que permiten acceder a sus valores a través de índices. Sin embargo, en ocasiones es importante poder nombrar elementos (en vez de utilizar posiciones) para acceder a valores y no queremos crear una clase ya que únicamente necesitamos un contenedor de valores y no comportamiento.

**Coffee** = collections.**NamedTuple**(‘**Coffee**’, (‘size’, ‘bean’, ‘price’))

**defget\_coffee**(coffee\_**type**):

**If** coffee\_**type** == ‘houseblend’:

**return** **Coffee**(‘large’, ‘premium’, 10)

El módulo collections también nos ofrece otros primitivos que tienen la labor de facilitarnos la creación y manipulación de colecciones en Python. Por ejemplo, Counter nos permite contar de manera eficiente ocurrencias en cualquier iterable; OrderedDict nos permite crear diccionarios que poseen un orden explícito; deque nos permite crear filas (para pilas podemos utilizar la lista).

En conclusión, el módulo collections es una gran fuente de utilerías que nos permiten escribir código más “pythonico” y más eficiente.

**Python comprehensions**

Las Comprehensions son constructos que nos permite generar una secuencia a partir de otra secuencia.

Existen tres tipos de comprehensions:

* List comprehensions

[**element** **for** **element** **in** element\_list **if** element\_meets\_condition]

* Dictionary comprehensions

{key: **element** **for** **element** **in** element\_list **if** element\_meets\_condition}

* Sets comprehensions

{**element** **for** **element** **in** element\_list **if** elements\_meets\_condition}

Ej: lista\_numeros = list(range(100))

>>> lista\_numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, …. 98, 99]

>>> pares = [numero for numero in lista\_numeros if numero % 2 == 0]

>>> pares = [0, 2, 4, 6, 8, 10, 12, 14, …. 92, 94, 96, 98]

>>> **student\_uid** = [1, 2, 3]

>>> **students** = ['juan', 'jose', 'lucho']

>>> nuevo\_students = {uid: student for uid, student in zip(**student\_uid**, **students**)}

> nuevo\_students = **{1: 'juan', 2: 'jose', 3: 'lucho'}** se creo un dicionari

->>> x = [1, 2, 3]

>>> y = [4, 5, 6]

>>> zipped = zip(x, y)

>>> list(zipped)= [(1, 4), (2, 5), (3, 6)]

>>> import random

>>> randon\_1 = []

>>> for i in range(10):

... randon\_1.append(random.randint(1, 3))

...

>>> randon\_1

[3, 2, 3, 1, 2, 3, 2, 1, 2, 3]

>>> no\_repetido = {number **for** number **in** randon\_1}

>>> no\_repetido = {1, 2, 3} elimino repetidos

**Búsquedas binarias**

Uno de los conceptos más importantes que debes entender en tu carrera dentro de la programación son los algoritmos. No son más que una secuencia de instrucciones para resolver un problema especifico.

Búsqueda binaria lo único que hace es tratar de encontrar un resultado en una lista ordenada de tal manera que podamos razonar. Si tenemos un elemento mayor que otro, podemos simplemente la mitad de la lista cada vez.

**sort** ordena la lista creada y **sorted** ordena una lista en una nueva sin modificar la original

La búsqueda binaria

Es posible aprovechar mejor la lista ordenada si somos inteligentes en nuestras comparaciones. En la búsqueda secuencial, cuando comparamos contra el primer ítem, hay a lo sumo n−1n−1 ítems restantes para verificar si el primer ítem no es el valor que estamos buscando. En lugar de buscar secuencialmente en la lista, una **búsqueda binaria** comenzará examinando el ítem central. Si ese ítem es el que estamos buscando, hemos terminado. Si no es el ítem correcto, podemos utilizar la naturaleza ordenada de la lista para eliminar la mitad de los ítems restantes. Si el ítem que buscamos es mayor que el ítem central, sabemos que toda la mitad inferior de la lista, así como el ítem central, se pueden ignorar de la consideración posterior. El ítem, si es que está en la lista, debe estar en la mitad superior.

Podemos entonces repetir el proceso con la mitad superior. Comenzar en el ítem central y compararlo con el valor que estamos buscando. Una vez más, o lo encontramos o dividimos la lista por la mitad, eliminando por tanto otra gran parte de nuestro espacio de búsqueda posible. La [Figura 3](https://interactivepython.org/runestone/static/pythoned/SortSearch/LaBusquedaBinaria.html#fig-binsearch) muestra cómo este algoritmo puede encontrar rápidamente el valor 54. La función completa se muestra en el [CodeLens 3](https://interactivepython.org/runestone/static/pythoned/SortSearch/LaBusquedaBinaria.html" \l "lst-binarysearchpy).

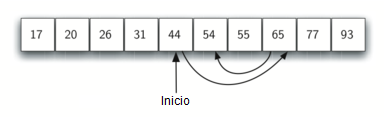


Figura 3: Búsqueda binaria en una lista ordenada de enteros

**PARA RECORDAR**

**if** \_\_name\_\_ == '\_\_main\_\_':

pass```

if \_\_**name\_\_** == “\_\_**main\_\_**”: es básicamente un entorno de script de nivel superior, especifica el intérprete que (‘Tengo la prioridad más alta para ejecutar primero’).

‘**main**’ es el nombre del ámbito en el que se ejecuta el código de nivel superior. El **name** un módulo se establece igual a ‘**main**’ cuando se lee desde la entrada estándar, un script o desde un indicador interactivo.

**Decoradores**

**Python** es un lenguaje que acepta **diversos** paradigmas como programación orientada a objetos y la programación funcional, siendo estos los temas de nuestro siguiente módulo.

Los **decoradores** son una función que envuelve a otra función para modificar o extender su comportamiento.

En Python las **funciones** son ciudadanos de primera clase, *first class citizen*, esto significan que las funciones pueden recibir funciones como **parámetros** y pueden **regresar** funciones. Los **decoradores** utilizan este concepto de manera fundamental.

DECORADORES EN PYTHON  
Los decoradores sirven para ejecutar lógica del código antes y/o después de otra función, esto nos ayuda a generar funciones y código que pueda ser reutilizado fácilmente sin hacer más extenso nuestro código. Hay que recordar que si se genera una función dentro de otra solo existiera en ese scope(dentro de la función padre), si se quiere invocar una función varias veces dentro de otras se tiene que generar de manera global.

**\*\***args y kwargs**\*\***

Básicamente lo que hacen es pasar tal cual los valores de de los argumentos que se pasan a la función args hace referencias a listas y kwargs a elementos de un diccionario (llave: valor)

\*\* args: \*\*

**deftest\_valor\_arg**(n\_arg, \*args):

print('primer valor normal: ', n\_arg)

For arg **in** args:

print('este es un valor de \*args: ',arg)

print(type(args))

**if**\_\_name\_\_ == '\_\_main\_\_':

test\_valor\_args('carlos','Karla','Paola','Elena')

* el tipo de valor y es una tupla
* solo poniendo argumentos divididos por comas los convierte

\*\*kuargs: \*\*

**deftest\_valor\_kwargs**(\*\*kwargs):

**if** kwargs **isnotNone**:

**for** key, value **in** kwargs.items():

print('%s == %s' %(key,value))

print(type(kwargs))

**if** \_\_name\_\_ == '\_\_main\_\_':

test\_valor\_kwargs(caricatura='batman')

* el valor que te da es un diccionario
* toma los valores en los extremos de un signo igual

**Este es un ejemplo usando los 2 en una función**

**deftest\_valor\_kwargs\_args**(\*args, \*\*kwargs):

print(type(kwargs))

print(kwargs)

print('----------')

print(type(args))

print(args)

**if** \_\_name\_\_ == '\_\_main\_\_':

test\_valor\_kwargs\_args('flash', 'batman', caricatura='batman', empresa = 'dc')

<https://recursospython.com/guias-y-manuales/argumentos-args-kwargs/>

**¿Qué es la programación orientada a objetos?**

La programación orientada a objetos es un paradigma de programación que otorga los medios para estructurar programas de tal manera que las propiedades y comportamientos estén envueltos en objetos individuales.

Para poder entender cómo modelar estos objetos tenemos que tener claros cuatro principios:

* Encapsulamiento.
* Abstracción
* Herencia
* Polimorfismo
* **Encapsulation**: Cada objeto tiene sus propias funciones, y datos sin afectar a otros, son lógica interna.
* **Abstraction**: El usuario podrá interactuar con el objeto sin necesidad de conocer toda la lógica del mismo.
* **Inheritance**: si se declara un método en una clase todas las subclases heredan ese método, es decir: si tu declaras un método “imprime” que ejecute un print en una clase, las subclases podrán usar el método imprime, sin necesidad de declararlo en cada una.
* **Polyphormism**: usando el ejemplo anterior, en cada subclase se puede modificar el método “imprime” por lo tal cada sub clase contara con un método imprime pero acorde a las necesidades de cada subclase.
* Encapsulamiento: Agrupar y esconder información.
* Abstracción: Elimina información adicional innecesaria.
* Herencia: Modelar jerarquía de objetos.
* Polimorfismo: Adaptar información en otros contextos.

Para declara una clase en Python utilizamos la keyword class, después de eso le damos el nombre. Una convención en Python es que todas las clases **empiecen con mayúscula** y se continua con **CamelCase**.

Un método fundamental es dunder init(\_\_init\_\_). Lo único que hace es inicializar la clase basado en los parámetros que le damos al momento de construir la clase.

self es una referencia a la clase. Es una forma internamente podemos acceder a las propiedades y métodos.

El **new**(dunder new). Se utiliza muy poco sin embargo tiene el proposito de usarse cuando se hacen subclases de de tipos inmutables como lo son las tuplas y los string.

Aqui les dejo una guia ligera!  
<https://rszalski.github.io/magicmethods/>

**new**(cls, args)  
Método exclusivo de las clases de nuevo estilo que se ejecuta antes que **init** y que se encarga de construir y devolver el objeto en sí. Es equivalente a los constructores de C++ o Java. Se trata de un método estático, es decir, que existe con independencia de las instancias de la clase: es un método de clase, no de objeto, y por lo tanto el primer parámetro no es self, sino la propia clase: cls.

Ej de \_\_int\_\_ y \_\_new\_\_

[https://medium.com/@\_seraph1/volviendo-a-lo-básico-poo-en-python-diferencia-entre-init-y-new-parte-10-3ec7632388d1](https://medium.com/@_seraph1/volviendo-a-lo-b%C3%A1sico-poo-en-python-diferencia-entre-init-y-new-parte-10-3ec7632388d1)

# Scopes and namespaces

En Python, un name, también conocido como identifier, es simplemente una forma de otorgarle un nombre a un objeto. Mediante el nombre, podemos acceder al objeto. Vamos a ver un ejemplo:

my\_var = 5id(my\_var) # 4561204416id(5) # 4561204416

En este caso, el identifier my\_var es simplemente una forma de acceder a un objeto en memoria (en este caso el espacio identificado por el número 4561204416). Es importante recordar que un name puede referirse a cualquier tipo de objeto (aún las funciones).

**defecho**(value):**return** value

**a** = echo

**a**(‘Billy’) # 3

Ahora que ya entendimos qué es un name podemos avanzar a los namespaces (espacios de nombres). Para ponerlo en palabras llanas, un namespace es simplemente un conjunto de names.

En Python, te puedes imaginar que existe una relación que liga a los nombres definidos con sus respectivos objetos (como un diccionario). Pueden coexistir varios namespaces en un momento dado, pero se encuentran completamente aislados. Por ejemplo, existe un namespace específico que agrupa todas las variables globales (por eso puedes utilizar varias funciones sin tener que importar los módulos correspondientes) y cada vez que declaramos una módulo o una función, dicho módulo o función tiene asignado otro namespace.

A pesar de existir una multiplicidad de namespaces, no siempre tenemos acceso a todos ellos desde un punto específico en nuestro programa. Es aquí donde el concepto de scope (campo de aplicación) entra en juego.

Scope es la parte del programa en el que podemos tener acceso a un namespace sin necesidad de prefijos.

En cualquier momento determinado, el programa tiene acceso a tres scopes:

* El scope dentro de una función (que tiene nombres locales)
* El scope del módulo (que tiene nombres globales)
* El scope raíz (que tiene los built-in names)

Cuando se solicita un objeto, Python busca primero el nombre en el scope local, luego en el global, y por último, en el raíz. Cuando anidamos una función dentro de otra función, su scope también queda anidado dentro del scope de la función padre.

**defouter\_function**(some\_local\_name):**definner\_function**(other\_local\_name):# Tiene acceso a la built-in function print y al nombre local some\_local\_name

print(some\_local\_name)

# También tiene acceso a su scope local

print(other\_local\_name)

Para poder manipular una variable que se encuentra fuera del scope local podemos utilizar los keywords global y nonlocal.

some\_var\_**in**\_other\_scope = 10**defsome\_function**():**global** some\_var\_**in**\_other\_scope

Some\_var\_**in**\_other\_scope += 1

En Python, la palabra clave **global** permite modificar la variable fuera del alcance actual. Se utiliza para crear una variable global y realizar cambios en la variable en un contexto local.

**Reglas de palabra clave global**

Las reglas básicas para la palabra clave **global** en Python son:

* Cuando creamos una variable dentro de una función, es local por defecto.
* Cuando definimos una variable fuera de una función, es global por defecto. No tienes que usar la palabra clave **global**.
* Utilizamos la palabra clave **global** para leer y escribir una variable global dentro de una función.
* El uso de **global** fuera de una función no tiene efecto

En lo referente a la palabra **nonlocal**, actúa de manera similar, solo que orientada a lo que sería un alcance de funciones y funciones anidadas, a continuación un ejemplo:

**Def method**():

**Def method2**():

# este método no tiene acceso a la

# variable value, por cuanto se usa

# nonlocal para poder acceder.

**nonlocal** value

value = 100

# Variable local.

value = 10

method2()

**Introducción a Click**

**Click** es un pequeño framework que nos permite crear aplicaciones de Línea de comandos. Tiene cuatro decoradores básicos:

* **@click\_group**: Agrupa una serie de comandos
* **@click\_command**: Aca definiremos todos los comandos de nuestra apliacion
* **@click\_argument**: Son parámetros necesarios
* **@click\_option**: Son parámetros opcionales

Click también realiza las conversiones de tipo por nosotros. Esta basado muy fuerte en decoradores.