Python - Clase 1

Created	@August 26, 2022 4:58 PM
Lesson Date	@August 30, 2022
Status	Complete
Type	Lesson Planning

Introducción:

Python es un lenguaje de programación interpretado y multiplataforma cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.

Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional.

Añadiría, como característica destacada, que se trata de un lenguaje de propósito general.

Algunas desventajas:

- Consumo de memoria.
- Errores durante la ejecución.
- Desarrollo móvil.
- Librerías de terceros no siempre del todo maduras.

Entornos virtuales:

Cuando trabajamos en distintos proyectos, no todos ellos requieren los mismos paquetes ni siquiera la misma versión de Python. La gestión de estas situaciones no es sencilla si
únicamente instalamos paquetes y manejamos configuraciones a nivel global. Es por ello que surge el concepto de entornos virtuales. Como su propio nombre
indica se trata de crear distintos entornos en función de las necesidades de cada proyecto, y esto nos permite establecer qué versión de Python usaremos y qué paquetes instalaremos.

Distintos tipos de entornos:

- Venv (entorno a utilizar)

```
# Crear el entorno env
python -m venv env

# Activación
env\Scripts\activate

# Desactivar entorno
deactivate
```

- Virtualenv
- Pyenv
- Conda

Primeros pasos:

Salida de datos:

1. La función `print()` es una función **integrada** imprime/envía un mensaje específico a la pantalla/ventana de consola.

2. Las funciones integradas, al contrario de las funciones definidas por el usuario, están siempre disponibles y no tienen que ser importadas.

3. Para llamar a una función (**invocación de función**), debe utilizarse el nombre de la función seguido de un paréntesis. Puedes pasar argumentos a una función colocándolos dentro de los paréntesis. Se Deben separar los argumentos con una coma, por ejemplo, `print("¡Hola,", "Mundo!")`. una función `print()` "vacía" imprime una línea vacía a la pantalla.

4. Las cadenas de Python están delimitadas por **comillas**, por ejemplo, `"Soy una cadena"`, o `'Yo soy una cadena, también'`.

5. Los programas de computadora son colecciones de **instrucciones**. Una instrucción es un comando para realizar una tarea específica cuando se ejecuta, por ejemplo, para imprimir un determinado mensaje en la pantalla.

6. En las cadenas de Python, la **barra diagonal inversa** (`\`) es un carácter especial que anuncia que el siguiente carácter tiene un significado diferente, por ejemplo, `\n` (el **carácter de nueva línea**) comienza una nueva línea de salida y `\t` como tabulador.

7. Los **argumentos posicionales** son aquellos cuyo significado viene dictado por su posición, por ejemplo, el segundo argumento se emite después del primero, el tercero se emite después del segundo, etc.

8. Los **argumentos de palabra clave** son aquellos cuyo significado no está dictado por su ubicación, sino por una palabra especial (palabra clave) que se utiliza para identificarlos.

9. Los parámetros `end` y `sep` se pueden usar para dar formato la salida de la función `print()`. El parámetro `sep` especifica el separador entre los argumentos emitidos (por ejemplo, `print("H", "E", "L", "L", "O", sep="-")`, mientras que el parámetro `end` especifica que imprimir al final de la declaración de impresión.

Operaciones básicas:

1. Una **expresión** es una combinación de valores (o variables, operadores, llamadas a funciones, aprenderás de ello pronto) las cuales son evaluadas y dan como resultado un valor, por ejemplo, `1 + 2`.

2. Los **operadores** son símbolos especiales o palabras clave que son capaces de operar en los valores y realizar operaciones matemáticas, por ejemplo, el `*` multiplica dos valores: `x * y`.

3. Los operadores aritméticos en Python: `+` (suma), `-` (resta), `*` (multiplicación), `/` (división clásica: regresa un flotante siempre), `%` (módulo: divide el operando izquierdo entre el operando derecho y regresa el residuo de la operación, por ejemplo, `5 % 2 = 1`), `**` (exponenciación: el operando izquierdo se eleva a la potencia del operando derecho, por ejemplo, `2 ** 3 = 2 * 2 * 2 = 8`), `//` (división entera: retorna el número resultado de la división, pero redondeado al número entero inferior más cercano, por ejemplo, `3 // 2.0 = 1.0`).

4. Un operador **unario** es un operador con solo un operando, por ejemplo, `-1`, o `+3`.

5. Un operador **binario** es un operador con dos operados, por ejemplo, `4 + 5`, o `12 % 5`.

6. Algunos operadores actúan antes que otros, a esto se le llama - **jerarquía de prioridades**:

- Unario `+` y `-` tienen la prioridad más alta.
- Después: `*`, después: `-`, `/`, y `%`, y después la prioridad más baja: binaria `+` y `-`.

7. Las sub-expresiones dentro de **paréntesis** siempre se calculan primero, por ejemplo, `15 - 1 * ( 5 *( 1 + 2 ) ) = 0`.

8. Los operadores de **exponenciación** utilizan **enlazado del lado derecho**, por ejemplo, `2 ** 2 ** 3 = 256`.

Variables:

1. Una **variable** es una ubicación nombrada reservada para almacenar valores en la memoria. Una variable es creada o inicializada automáticamente cuando se le asigna un valor por primera vez. (2.1.4.1)

2. Cada variable debe de tener un nombre único - un **identificador**. Un nombre válido debe ser aquel que no contiene espacios, debe comenzar con un guion bajo (`_`), o una letra, y no puede ser una palabra reservada de Python. El primer carácter puede estar seguido de guiones bajos, letras, y dígitos. Las variables en Python son sensibles a mayúsculas y minúsculas. (2.1.4.1)

3. Python es un lenguaje **de tipo dinámico**, lo que significa que no se necesita *declarar* variables en él. (2.1.4.3) Para asignar valores a las variables, se utiliza simplemente el operador de asignación, es decir el signo de igual (`=`) por ejemplo, `var = 1`.

4. Se les puede asignar valores nuevos a variables ya existentes utilizando el operador de asignación o un operador abreviado, por ejemplo (2.1.4.5):

```
var = 2
print(var)

var = 3
print(var)

var += 1
print(var)
```

5. Se puede combinar texto con variables empleado el operador `+`, y utilizar la función `print()` para mostrar o imprimir los resultados, por ejemplo: (2.1.4.4)

```
var = "007"
print("Agente " + var)
```

Entrada de datos:

La función `print()` **envía datos a la consola**, mientras que la función `input()` **obtiene datos de la consola**.

2. La función `input()` viene con un parámetro inicial: **un mensaje de tipo cadena para el usuario**. Permite escribir un mensaje antes de la entrada del usuario, por ejemplo:

```
name = input("Ingresa tu nombre: ")
print("Hola, " + name + ". ¡Un gusto conocerte!")
```

3. Cuando la función `input()` es llamada o invocada, el flujo del programa se detiene, el símbolo del cursor se mantiene parpadeando (le está indicando al usuario que tome acción ya que la consola está en modo de entrada) hasta que el usuario haya ingresado un dato y/o haya presionado la tecla *Enter*.

```
name = input("Ingresa tu nombre: ")
print("Hola, " + name + ". ¡Un gusto conocerte!")

print("\nPresiona la tecla Enter para finalizar el programa.")

input()
print("FIN.")
```

4. El resultado de la función `input()` es una cadena. Se pueden unir cadenas unas con otras a través del operador de concatenación (`+`). Observa el siguiente código:

```
num_1 = input("Ingresa el primer número: ") # Ingresa 12
num_2 = input("Ingresa el segundo número: ") # Ingresa 21

print(num_1 + num_2) el programa retorna 1221
```

5. También se pueden multiplicar (`*` - replicación) cadenas, por ejemplo:

```
my_input = input("Ingresa algo: ") # Ejemplo: hola
print(my_input * 3) # Salida esperada: holaholahola
```

Tipos de datos:

Cadenas

Las cadenas en Python o `strings` son un tipo inmutable que permite almacenar secuencias de caracteres. Para crear una, es necesario incluir el texto entre comillas dobles o simples.

```
s = "Esto es una cadena"
e = 'Esto es otra cadena'
a = ''
print(s)       #Esto es una cadena
print(type(s)) #<class 'str'>
```

Tal vez queramos declarar una cadena que contenga variables en su interior, como números o incluso otras cadenas. Una forma de hacerlo sería concatenando la cadena que queremos con otra usando el operador `+`. Nótese que `str()` convierte en `string` lo que se pasa como parámetro.

```
x = str(10.4)
print(x)       #10.4
print(type(x)) #<class 'str'>
```

cantidad de caracteres con `len()`

Metodos:

Algunos de los métodos de la clase `string`.

`capitalize()`

El método `capitalize()` se aplica sobre una cadena y la devuelve con su primera letra en mayúscula.

```
s = "mi cadena"
print(s.capitalize()) #Mi cadena
```

`lower()`

El método `lower()` convierte todos los caracteres alfabéticos en minúscula.

```
s = "MI CADENA"
print(s.lower()) #mi cadena
```

`swapcase()`

El método `swapcase()` convierte los caracteres alfabéticos con mayúsculas en minúsculas y viceversa.

```
s = "mI cAdEnA"
print(s.swapcase()) #Mi CaDeNa
```

`upper()`

El método `upper()` convierte todos los caracteres alfabéticos en mayúsculas.

```
s = "mi cadena"
print(s.upper())
```

`count(<sub>[, <start>[, <end>]])`

El método `count()` permite contar las veces que otra cadena se encuentra dentro de la primera. Permite también dos parámetros opcionales que indican donde empezar y acabar de buscar.

```
s = "el bello cuello "
print(s.count("llo")) #2
```

`isalnum()`

El método `isalnum()` devuelve `True` si la cadena esta formada únicamente por caracteres alfanuméricos, `False` de lo contrario. Caracteres como `@` o `&` no son alfanumericos.

```
s = "correo@dominio.com"
print(s.isalnum())
```

`isalpha()`

El método `isalpha()` devuelve `True` si todos los caracteres son alfabéticos, `False` de lo contrario.

```
s = "abcdefg"
print(s.isalpha())
```

`strip([<chars>])`

El método `strip()` elimina a la izquierda y derecha el carácter que se le introduce. Si se llama sin parámetros elimina los espacios. Muy útil para limpiar cadenas.

```
s = "  abc  "
print(s.strip()) #abc
```

`zfill(<width>)`

El método `zfill()` rellena la cadena con ceros a la izquierda hasta llegar a la longitud pasada como parámetro.

```
s = "123"
print(s.zfill(5)) #00123
```

`join(<iterable>)`

El método `join()` devuelve la primera cadena unida a cada uno de los elementos de la lista que se le pasa como parámetro.

```
s = " y ".join(["1", "2", "3"])
print(s) #1 y 2 y 3
```

`split(sep=None, maxsplit=-1)`

El método `split()` divide una cadena en subcadenas y las devuelve almacenadas en una lista. La división es realizada de acuerdo a el primer parámetro, y el segundo parámetro indica el número máximo de divisiones a realizar.

```
s = "Python,Java,C"
print(s.split(",")) #['Python', 'Java', 'C']
```

Listas

Las listas en Python son un tipo de dato que permite almacenar datos de cualquier tipo.

Algunas propiedades de las listas:lista = [1, 2, 3, 4]

- Son **ordenadas**, mantienen el orden en el que han sido definidas
- Pueden ser formadas por tipos **arbitrarios**
- Pueden ser **indexadas** con `[i]`.
- Se pueden **anidar**, es decir, meter una dentro de la otra.
- Son **mutables**, ya que sus elementos pueden ser modificados.
- Son **dinámicas**, ya que se pueden añadir o eliminar elementos.

```
lista = [1, 2, 3, 4]
lista = list("1234")
lista = [1, "Hola", 3.67, [1, 2, 3]]
```

Acceder y modificar listas

Si tenemos una lista `a` con 3 elementos almacenados en ella, podemos acceder a los mismos usando corchetes y un índice, que va desde `0` a `n-1` siendo `n` el tamaño de la lista.

```
a = [90, "Python", 3.87]
print(a[0]) #90
print(a[1]) #Python
print(a[2]) #3.87
```

Se puede también acceder al último elemento usando el índice `[-1]`.

```
a = [90, "Python", 3.87]
print(a[-1]) #3.87
```

De la misma manera, al igual que `[-1]` es el último elemento, podemos acceder a `[-2]` que será el penúltimo.

`print(a[-2]) #Python`

Y si queremos modificar un elemento de la lista, basta con asignar con el operador `=` el nuevo valor.

```
a[2] = 1
print(a) #[90, 'Python', 1]
```

Un elemento puede ser eliminado con diferentes métodos como veremos a continuación, o con `del` y la lista con el índice a eliminar.

```
l = [1, 2, 3, 4, 5]
del l[1]
print(l) #[1, 3, 4, 5]
```

También podemos tener **listas anidadas**, es decir, una lista dentro de otra. Incluso podemos tener una lista dentro de otra lista y a su vez dentro de otra lista. Para acceder a sus elementos sólo tenemos que usar `[]` tantas veces como niveles de anidado tengamos.

```
x = [1, 2, 3, ['p', 'q', [5, 6, 7]]]
print(x[3][0])    #p
print(x[3][2][0]) #5
print(x[3][2][2]) #7
```

También es posible crear sublistas más pequeñas de una más grande. Para ello debemos de usar `:` entre corchetes, indicando a la izquierda el valor de inicio, y a la izquierda el valor final que no está incluido. Por lo tanto `[0:2]` creará una lista con los elementos `[0]` y `[1]` de la original.

```
l = [1, 2, 3, 4, 5, 6]
print(l[0:2]) #[1, 2]
print(l[2:6]) #[3, 4, 5, 6]
```

Y de la misma manera podemos modificar múltiples valores de la lista a la vez usando `:`.

```
l = [1, 2, 3, 4, 5, 6]
l[0:3] = [0, 0, 0]
print(l) #[0, 0, 0, 4, 5, 6]
```

Iterar listas

En Python es muy fácil iterar una lista, mucho más que en otros lenguajes de programación.

```
lista = [5, 9, 10]
for l in lista:
    print(l)
#5
#9
#10
```

Métodos listas

`append(<obj>)`

El método `append()` añade un elemento al final de la lista.

```
l = [1, 2]
l.append(3)
print(l) #[1, 2, 3]
```

`extend(<iterable>)`

El método `extend()` permite añadir una lista a la lista inicial.

```
l = [1, 2]
l.extend([3, 4])
print(l) #[1, 2, 3, 4]
```

`insert(<index>, <obj>)`

El método `insert()` añade un elemento en una posición o índice determinado.

```
l = [1, 3]
l.insert(1, 2)
print(l) #[1, 2, 3]
```

`remove(<obj>)`

El método `remove()` recibe como argumento un objeto y lo borra de la lista.

```
l = [1, 2, 3]
l.remove(3)
print(l) #[1, 2]
```

`pop(index=-1)`

El método `pop()` elimina por defecto el último elemento de la lista, pero si se pasa como parámetro un `indice` permite borrar elementos diferentes al último.

```
l = [1, 2, 3]
l.pop()
print(l) #[1, 2]
```

`reverse()`

El método `reverse()` inverte el órden de la lista.

```
l = [1, 2, 3]
l.reverse()
print(l) #[3, 2, 1]
```

`sort()`

El método `sort()` ordena los elementos de menos a mayor por defecto.

```
l = [3, 1, 2]
l.sort()
print(l) #[1, 2, 3]
```

Y también permite ordenar de mayor a menor si se pasa como parámetro `reverse=True`.

```
l = [3, 1, 2]
l.sort(reverse=True)
print(l) #[3, 2, 1]
```

`index(<obj>[,index])`

El método `index()` recibe como parámetro un objeto y devuelve el índice de su primera aparición. Como hemos visto en otras ocasiones, el índice del primer elemento es el `0`.

```
l = ["Periphery", "Intervals", "Monuments"]
print(l.index("Intervals"))
```

También permite introducir un parámetro opcional que representa el índice desde el que comenzar la búsqueda del objeto. Es como si ignorara todo lo que hay antes de ese índice para la búsqueda, en este caso el `4`.

```
l = [1, 1, 1, 1, 2, 1, 4, 5]
print(l.index(1, 4)) #5
```

Tupla (tuple)

Las tuplas en Python son un tipo o estructura de datos que permite almacenar datos de una manera muy parecida a las listas, con la salvedad de que son inmutables.

Crear tupla Python

Las tuplas en Python o `tuples` son muy similares a las listas, pero con dos diferencias. Son **inmutables**, lo que significa que no pueden ser modificadas una vez declaradas, y en vez de inicializarse con corchetes se hace con `()`. Dependiendo de lo que queramos hacer, **las tuplas pueden ser más rápidas**.

```
tupla = (1, 2, 3)
print(tupla) #(1, 2, 3)
```

También pueden declararse sin `()`, separando por `,` todos sus elementos.

```
tupla = 1, 2, 3
print(type(tupla)) #<class 'tuple'>
print(tupla)       #(1, 2, 3)
```

Operaciones con tuplas

Como hemos comentado, las tuplas son tipos **inmutables**, lo que significa que una vez asignado su valor, no puede ser modificado. Si se intenta, tendremos un `TypeError`.

```
tupla = (1, 2, 3)
#tupla[0] = 5 # Error! TypeError
```

Al igual que las listas, las tuplas también pueden ser anidadas.

```
tupla = 1, 2, ('a', 'b'), 3
print(tupla)       #(1, 2, ('a', 'b'), 3)
print(tupla[2][0]) #a
```

Y también es posible convertir una lista en tupla haciendo uso de al función `tuple()`.

```
lista = [1, 2, 3]
tupla = tuple(lista)
print(type(tupla)) #<class 'tuple'>
print(tupla)       #(1, 2, 3)
```

Se puede **iterar** una tupla de la misma forma que se hacía con las listas.

```
tupla = [1, 2, 3]
for t in tupla:
    print(t) #1, 2, 3
```

Y se puede también asignar el valor de una tupla con `n` elementos a `n` variables.

```
l = (1, 2, 3)
x, y, z = l
print(x, y, z) #1 2 3
```

Aunque tal vez no tenga mucho sentido a nivel práctico, es posible crear una tupla de un solo elemento. Para ello debes usar `,` antes del paréntesis, porque de lo contrario `(2)` sería interpretado como `int`.

```
tupla = (2,)
print(type(tupla)) #<class 'tuple'>
```

Métodos tuplas

`count(<obj>)`

El método `count()` cuenta el número de veces que el objeto pasado como parámetro se ha encontrado en la lista.

```
l = [1, 1, 1, 3, 5]
print(l.count(1)) #3
```

`index(<obj>[,index])`

El método `index()` busca el objeto que se le pasa como parámetro y devuelve el índice en el que se ha encontrado.

```
l = [7, 7, 7, 3, 5]
print(l.index(5)) #4
```

En el caso de no encontrarse, se devuelve un `ValueError`.

```
l = [7, 7, 7, 3, 5]
#print(l.index(35)) #Error! ValueErrorEl método index() también acepta un segundo parámetro opcional, que indica a partir de que índice emp
```

Diccionario

Los diccionarios en Python son una estructura de datos que permite almacenar su contenido en forma de llave y valor.

Crear diccionario Python

Un diccionario en Python es una colección de elementos, donde cada uno tiene una llave `key` y un valor `value`. Los diccionarios se pueden crear con paréntesis `{}` separando con una coma cada par `key: value`. En el siguiente ejemplo tenemos tres `keys` que son el nombre, la edad y el documento.

```
d1 = {
  "Nombre": "Sara",
  "Edad": 27,
  "Documento": 1003882
}
print(d1)
#{'Nombre': 'Sara', 'Edad': 27, 'Documento': 1003882}
```

Otra forma equivalente de crear un diccionario en Python es usando `dict()` e introduciendo los pares `key: value` entre paréntesis.

```
d2 = dict([
      ('Nombre', 'Sara'),
      ('Edad', 27),
      ('Documento', 1003882),
])
print(d2)
#{'Nombre': 'Sara', 'Edad': '27', 'Documento': '1003882'}
```

También es posible usar el constructor `dict()` para crear un diccionario.

```
d3 = dict(Nombre='Sara',
          Edad=27,
          Documento=1003882)
print(d3)
#{'Nombre': 'Sara', 'Edad': 27, 'Documento': 1003882}
```

Algunas propiedades de los diccionario en Python son las siguientes:

- Son **dinámicos**, pueden crecer o decrecer, se pueden añadir o eliminar elementos.
- Son **indexados**, los elementos del diccionario son accesibles a través del `key`.
- Y son **anidados**, un diccionario puede contener a otro diccionario en su campo `value`.

Acceder y modificar elementos

Se puede acceder a sus elementos con `[]` o también con la función `get()`

```
print(d1['Nombre'])     #Sara
print(d1.get('Nombre')) #Sara
```

Para modificar un elemento basta con usar `[]` con el nombre del `key` y asignar el valor que queremos.

```
d1['Nombre'] = "Laura"
print(d1)
#{'Nombre': Laura', 'Edad': 27, 'Documento': 1003882}
```

Si el `key` al que accedemos no existe, se añade automáticamente.

```
d1['Direccion'] = "Calle 123"
print(d1)
#{'Nombre': 'Laura', 'Edad': 27, 'Documento': 1003882, 'Direccion': 'Calle 123'}
```

Iterar diccionario

Los diccionarios se pueden iterar de manera muy similar a las listas u otras estructuras de datos. Para imprimir los `key`.

```
# Imprime los key del diccionario
for x in d1:
    print(x)
#Nombre
#Edad
#Documento
#Direccion
```

Se puede imprimir también solo el `value`.

```
# Impre los value del diccionario
for x in d1:
    print(d1[x])
#Laura
#27
#1003882
#Calle 123
```

O si queremos imprimir el `key` y el `value` a la vez.

```
# Imprime los key y value del diccionario
for x, y in d1.items():
    print(x, y)
#Nombre Laura
#Edad 27
#Documento 1003882
#Direccion Calle 123
```

Diccionarios anidados

Los diccionarios en Python pueden contener uno dentro de otro. Podemos ver como los valores anidado uno y dos del diccionario `d` contienen a su vez otro diccionario.

```
anidado1 = {"a": 1, "b": 2}
anidado2 = {"a": 1, "b": 2}
d = {
  "anidado1" : anidado1,
  "anidado2" : anidado2
}
print(d)
#{'anidado1': {'a': 1, 'b': 2}, 'anidado2': {'a': 1, 'b': 2}}
```

Métodos diccionarios Python

`clear()`

El método `clear()` elimina todo el contenido del diccionario.

```
d = {'a': 1, 'b': 2}
d.clear()
print(d) #{}
```

`get(<key>[,<default>])`

El método `get()` nos permite consultar el `value` para un `key` determinado. El segundo parámetro es opcional, y en el caso de proporcionarlo es el valor a devolver si no se encuentra la `key`.

```
d = {'a': 1, 'b': 2}
print(d.get('a')) #1
print(d.get('z', 'No encontrado')) #No encontrado
```

`items()`

El método `items()` devuelve una lista con los `keys` y `values` del diccionario. Si se convierte en `list` se puede indexar como si de una lista normal se tratase, siendo los primeros elementos las `key` y los segundos los `value`.

```
d = {'a': 1, 'b': 2}
it = d.items()
print(it)             #dict_items([('a', 1), ('b', 2)])
print(list(it))       #[('a', 1), ('b', 2)]
print(list(it)[0][0]) #a
```

`keys()`

El método `keys()` devuelve una lista con todas las `keys` del diccionario.

```
d = {'a': 1, 'b': 2}
k = d.keys()
print(k)       #dict_keys(['a', 'b'])
print(list(k)) #['a', 'b']
```

`values()`

El método `values()` devuelve una lista con todos los `values` o valores del diccionario.

```
d = {'a': 1, 'b': 2}
print(list(d.values())) #[1, 2]
```

`pop(<key>[,<default>])`

El método `pop()` busca y elimina la `key` que se pasa como parámetro y devuelve su valor asociado. Daría un error si se intenta eliminar una `key` que no existe.

```
d = {'a': 1, 'b': 2}
d.pop('a')
print(d) #{'b': 2}
```

También se puede pasar un segundo parámetro que es el valor a devolver si la `key` no se ha encontrado. En este caso si no se encuentra no habría error.

```
d = {'a': 1, 'b': 2}
d.pop('c', -1)
print(d) #{'a': 1, 'b': 2}
```

`popitem()`

El método `popitem()` elimina de manera aleatoria un elemento del diccionario.

```
d = {'a': 1, 'b': 2}
d.popitem()
print(d)
#{'a': 1}
```

`update(<obj>)`

El método `update()` se llama sobre un diccionario y tiene como entrada otro diccionario. Los `value` son actualizados y si alguna `key` del nuevo diccionario no esta, es añadida.

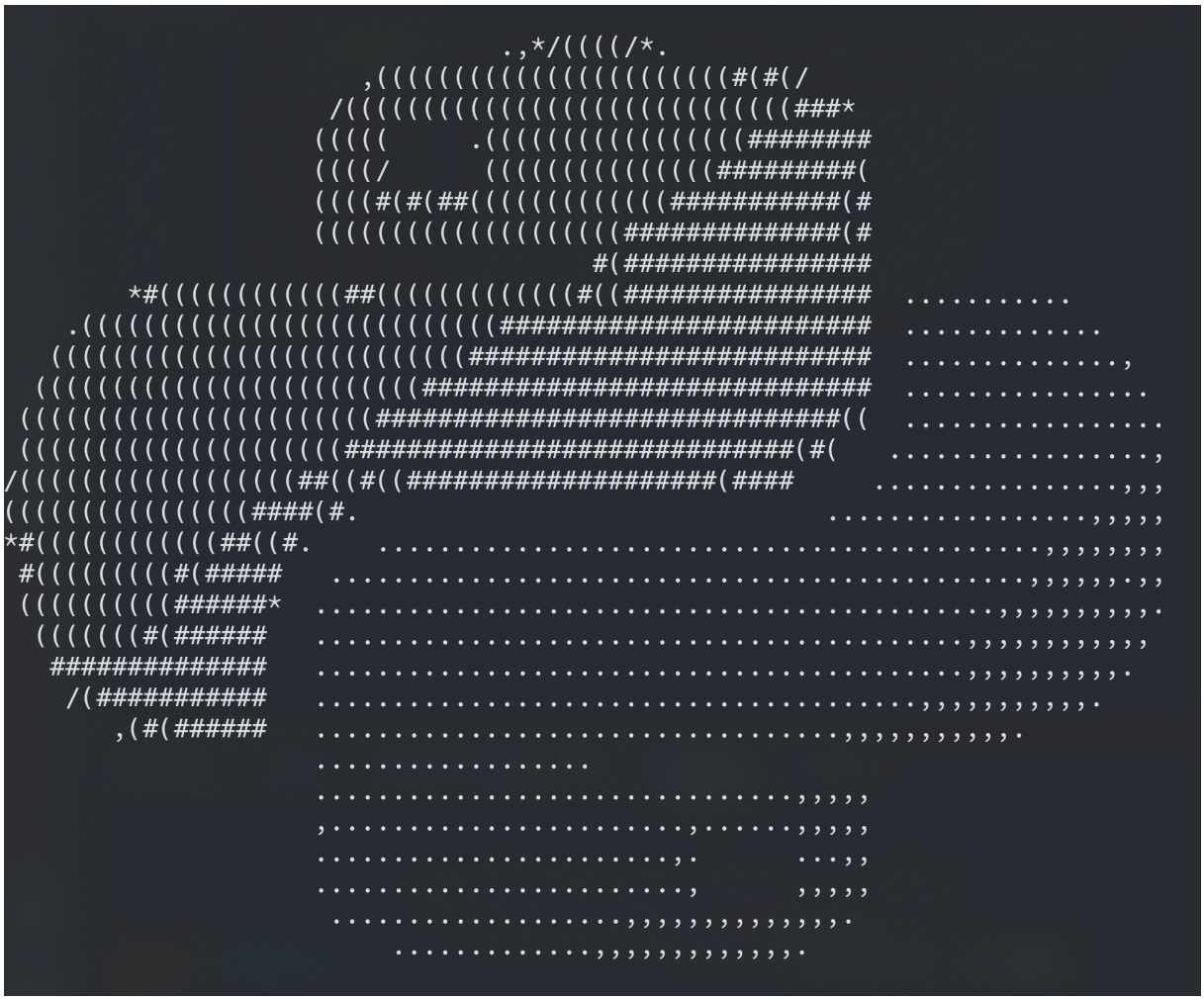
```
d1 = {'a': 1, 'b': 2}
d2 = {'a': 0, 'd': 400}
d1.update(d2)
print(d1)
#{'a': 0, 'b': 2, 'd': 400}
```

Ejercicios:

1. Idear la manera de realizar la siguiente salida (a, b y c son variables).:

   ```
   a       b       c
   2.4     -3.21   47.8
   ```

2. Mostrar por pantalla alguna imagen o dibujo.



3. Escriba un programa que lea 1 palabra (ingresadas por el usuario), calcule la longitud de cada una de ellas y las despliegue junto con su longitud indicada con asteriscos. Ejemplo:

   Árbol ******
   Celular *******
   Uno ***

4. Crear un diccionario del ejercicio anterior.