

Contenido

- ▼ [1 Fundamentos del lenguaje](#)
 - [1.1 Python](#)
 - [1.2 Instalando Python](#)
 - [1.3 La consola de Python como una calculadora básica](#)
 - ▼ [1.4 Variables y tipos de datos](#)
 - [1.4.1 Variables](#)
 - [1.4.2 Enteros](#)
 - [1.4.3 De coma flotante](#)
 - [1.4.4 Cadenas de caracteres](#)
 - [1.4.5 Booleanos](#)
 - [1.4.6 Listas](#)
 - [1.4.7 Tuplas](#)
 - [1.4.8 Diccionarios](#)
 - [1.5 Operadores relacionales y lógicos](#)
 - ▼ [1.6 Control de flujo](#)
 - [1.6.1 Condicional `if-elif-else`](#)
 - [1.6.2 Bucle `for`](#)
 - [1.6.3 Bucle `while`](#)
 - ▼ [1.7 Funciones](#)
 - [1.7.1 Funciones nativas de Python \(`built-in` \)](#)
 - [1.7.2 Funciones definidas por el usuario](#)
 - [1.7.3 Funciones con una cantidad de parámetros indeterminada](#)
 - [1.7.4 Funciones y los argumentos con nombre](#)
 - [1.8 Ejercicios](#)

1 Fundamentos del lenguaje

1.1 Python

Python es un lenguaje de programación, interpretado, de alto nivel y propósito general, además de ser un proyecto libre y de código abierto, con una comunidad enorme implicada en el desarrollo y mantenimiento de librerías que hacen posible el *multidominio* actual de Python.

Dada su concepción como lenguaje de propósito general, Python es utilizado en una diversidad de aplicaciones, desde desarrollo web, encriptación, análisis de datos, procesamiento de imágenes, aprendizaje automático, computación simbólica, etc.

Las características de este lenguaje le hacen propicio para el prototipado de aplicaciones, dado que es muy sencillo y rápido revisar y modificar el código desarrollado. Otra característica muy notable de Python es su sintaxis simple y fácil de aprender, lo cual ayuda al momento de introducirse en el desarrollo de algoritmos o el mundo propio de la programación de computadoras.

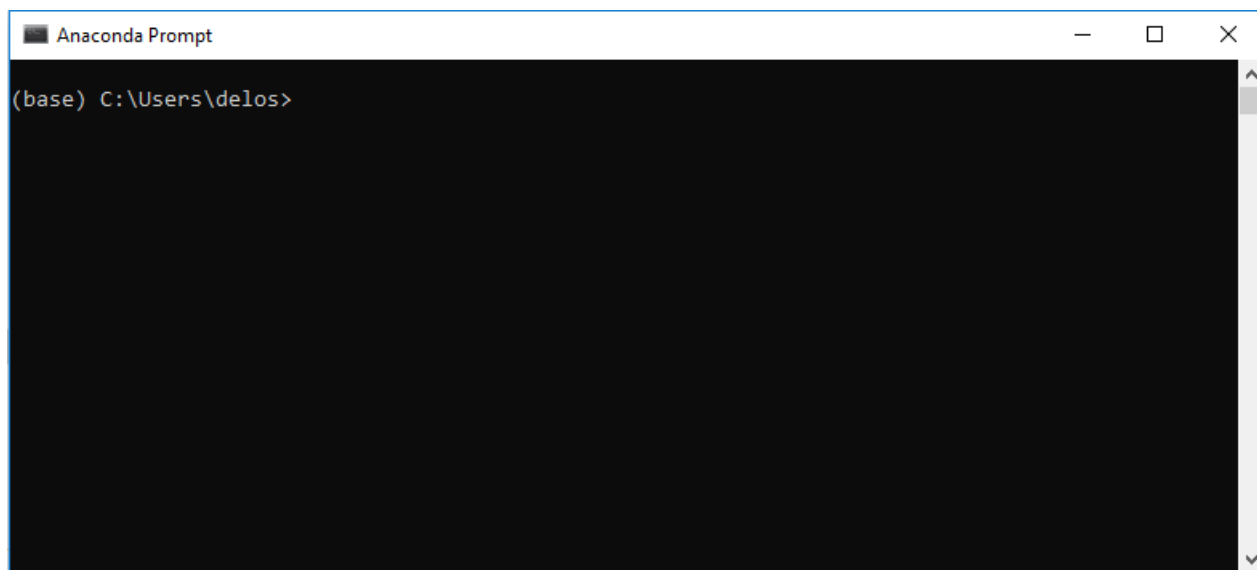
1.2 Instalando Python

En estos apuntes se utilizará la distribución Anaconda de Python, la cual contiene el intérprete y las librerías del *core*, pero además incluye la mayoría de librerías utilizadas para el desarrollo de aplicaciones de corte técnico-científico.

La descarga de Anaconda puede realizarla desde el sitio <https://www.anaconda.com/download/> (<https://www.anaconda.com/download/>), selecciona el paquete de descarga conforme al sistema operativo (Windows, macOS o Linux) así como la arquitectura de su PC. La instalación suele ser muy sencilla, puede seguir las instrucciones dadas en el *How to install ANACONDA* de la misma página.

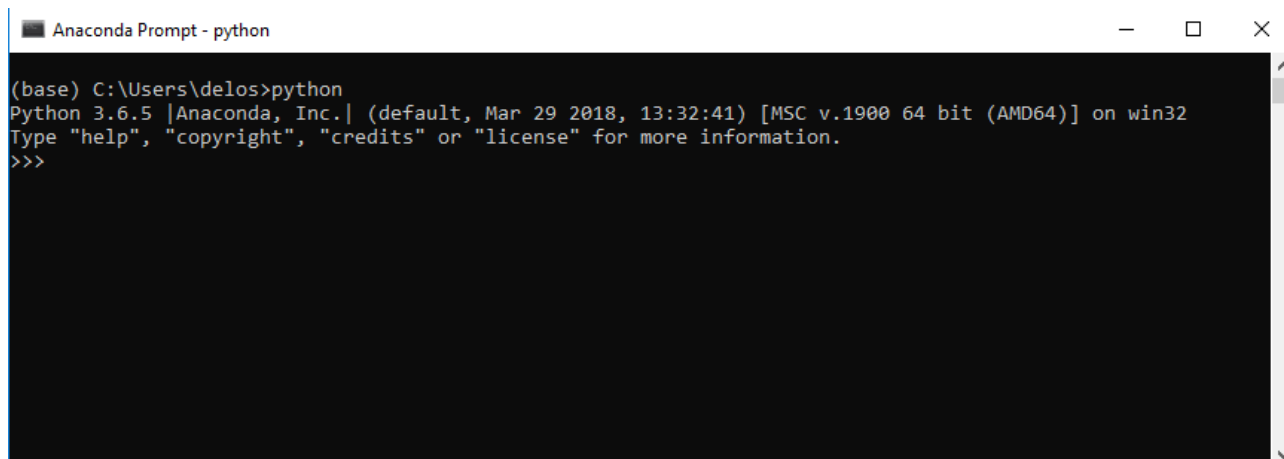
1.3 La consola de Python como una calculadora básica

Una vez instalado Anaconda puede testear la correcta instalación abriendo `Anaconda Prompt`, la cual es una consola desde la cual puede ejecutar algunos Scripts que complementan el funcionamiento de Python. Puede buscar esta consola en la carpeta de instalación correspondiente. Cuando la ejecute observará una consola como la mostrada en la figura.



```
(base) C:\Users\delos>
```

Dentro de esta consola escriba `python` y presione la tecla **Enter**, al hacer esto observará que la consola cambia el prompt compuesto de un directorio por tres signos de *mayor que* tal como se puede verificar en la figura.



```
(base) C:\Users\delos>python
Python 3.6.5 [Anaconda, Inc.] (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

A partir de este momento puede ingresar código Python y teclear **Enter** para ejecutar la instrucción y la consola le devolverá lo que resulte de esto. Por ejemplo, si escribe un número cualesquiera y presiona enter, la consola le devolverá justamente el mismo número:

```
In [1]: 1000
```

```
Out[1]: 1000
```

Puede ejecutar una simple suma aritmética:

```
In [2]: 100 + 200
```

```
Out[2]: 300
```

O una resta:

```
In [3]: 550 - 650
```

```
Out[3]: -100
```

Naturalmente Python maneja sin complicaciones las cantidades negativas. Una multiplicación la realiza con el operador `*`:

```
In [5]: 50*25
```

```
Out[5]: 1250
```

Para las divisiones utiliza el operador `/`:

```
In [6]: 1/2
```

```
Out[6]: 0.5
```

Puede elevar a una potencia utilizando como operador el doble asterisco:

```
In [7]: 13**2
```

```
Out[7]: 169
```

Inclusive existe la posibilidad de definir números complejos y realizar operaciones con ellos:

```
In [8]: 5 + 2j
```

```
Out[8]: (5+2j)
```

```
In [9]: (5 + 2j) - (10 + 7j)
```

```
Out[9]: (-5-5j)
```

```
In [10]: (5 + 2j)*(10 + 7j)
```

```
Out[10]: (36+55j)
```

Puede ampliar la capacidad de las funcionalidades *built-in* de Python si importa alguna librería, como `math`, pero claro, eso será un tema a tratar con posterioridad.

1.4 Variables y tipos de datos

Al ser un lenguaje de alto nivel, Python dispone de los tipos de datos elementales en cualquier lenguaje de programación, pero además incluye estructuras de datos muy *avanzadas* y con altas prestaciones que facilitan en muchos aspectos la tarea del programador.

Python es un lenguaje de tipado dinámico en el que no hace falta declarar el tipo de dato que asignará a una variable, de igual manera una variable puede cambiar de tipo conforme la ejecución del programa, por ello se debe tener cuidado con la sintaxis para definir cada tipo de dato.

1.4.1 Variables

Las variables son referencias a los objetos de Python, son creadas por asignación mediante el signo `=`, por ejemplo:

```
In [13]: a = 2  
         b = 10  
         a + b
```

```
Out[13]: 12
```

El nombre de una variable puede constar de una combinación de caracteres alfanuméricos y el guión bajo, siempre y cuando el primer carácter no sea un dígito. Además, en Python los nombres de variables son *case sensitive*, es decir, se distingue entre mayúsculas y minúsculas.

```
In [17]: D = 177.8  
         d = 95
```

```
In [18]: print(D)
```

```
177.8
```

```
In [19]: print(d)
```

```
95
```

Existen algunas palabras reservadas del lenguaje que no puede utilizar como nombre de variable, puede verificar cuáles son estas palabras tecleando lo siguiente:

```
In [21]: import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

1.4.2 Enteros

1.4.3 De coma flotante

1.4.4 Cadenas de caracteres

Las cadenas de caracteres (denominadas habitualmente y de manera indistinta como *strings* es un tipo de dato que contiene una secuencia de símbolos, mismos que pueden ser alfanúmericos hasta cualquier otro símbolo propio de un sistema de escritura. En Python los strings se definen entre comillas dobles o simples:

```
In [22]: "esta es una cadena de caracteres"
```

```
Out[22]: 'esta es una cadena de caracteres'
```

```
In [23]: 'esta también'
```

```
Out[23]: 'esta también'
```

Puede concatenar dos strings utilizando el operador `+` :

```
In [24]: "Hola" + "mundo"
```

```
Out[24]: 'Holamundo'
```

Notará que Python por sí mismo no sabe que estamos uniendo dos palabras y que entre ellas debería haber un espacio para su correcta lectura, evidentemente este tipo de cuestiones son las que el programador debe tomar en cuenta al escribir un código.

Una cadena de caracteres es lo que en Python se conoce como *iterable*, es decir, una secuencia de elementos agrupados a los cuales se puede acceder de manera individual mediante indexación. Por ejemplo, sea `nombre` una cadena de caracteres dada por:

```
In [25]: nombre="Catalina"
```

Puede acceder a cada una de las letras que componen dicha cadena mediante la notación `iter[pos]` , donde `iter` es el nombre del iterable y `pos` la posición en que se encuentra el caracter al cual se desea acceder, siendo 0 para la primera letra, 1 para la segunda y así de manera consecutiva. Por ejemplo:

```
In [26]: nombre[0]
```

```
Out[26]: 'C'
```

```
In [28]: nombre[4]
```

```
Out[28]: 'l'
```

```
In [29]: nombre[2]
```

```
Out[29]: 't'
```

Al último elemento, sin importar la longitud de la cadena, se accede con el índice -1:

```
In [30]: nombre[-1]
```

```
Out[30]: 'a'
```

1.4.5 Booleanos

1.4.6 Listas

Las listas son estructuras de datos que pueden almacenar cualquier otro tipo de dato, inclusive una lista puede contener otra lista, además, la cantidad de elementos de una lista se puede modificar removiendo o añadiendo elementos. Para definir una lista se utilizan los corchetes, dentro de estos se colocan todos los elementos separados por comas:

```
In [36]: calificaciones = [10,9,8,7.5,9]
nombres = ["Ana","Juan","Sofía","Pablo","Tania"]
mezcla = [True, 10.5, "abc", [0,1,1]]
```

Las listas son iterables y por tanto se puede acceder a sus elementos mediante indexación:

```
In [37]: nombres[2]
```

```
Out[37]: 'Sofía'
```

```
In [38]: nombres[-1]
```

```
Out[38]: 'Tania'
```

Se tiene la posibilidad de agregar elementos a una lista mediante el método `append` :

```
In [39]: nombres.append("Antonio")
nombres.append("Ximena")
print(nombres)
```

```
['Ana', 'Juan', 'Sofía', 'Pablo', 'Tania', 'Antonio', 'Ximena']
```

El método `remove` elimina un elemento de una lista:

```
In [40]: nombres.remove("Ana")
print(nombres)
```

```
['Juan', 'Sofía', 'Pablo', 'Tania', 'Antonio', 'Ximena']
```

Sí el valor pasado al método `remove` no existe, Python devolverá un `ValueError` :

```
In [41]: nombres.remove("Jorge")
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-41-d983d2559e2f> in <module>()
----> 1 nombres.remove("Jorge")

ValueError: list.remove(x): x not in list
```

1.4.7 Tuplas

Las tuplas son secuencias de elementos similares a las listas, la diferencia principal es que las tuplas no pueden ser modificadas directamente, es decir, una tupla no dispone de los métodos como `append` o `insert` que modifican los elementos de una lista.

Para definir una tupla, los elementos se separan con comas y se encierran entre paréntesis.

```
In [42]: colores=("Azul","Verde","Rojo","Amarillo","Blanco","Negro","Gris")
```

Las tuplas al ser *iterables* pueden accederse mediante la notación de corchetes e índice.

```
In [44]: colores[0]
```

```
Out[44]: 'Azul'
```

```
In [45]: colores[-1]
```

```
Out[45]: 'Gris'
```

```
In [46]: colores[3]
```

```
Out[46]: 'Amarillo'
```

Si intentamos modificar alguno de los elementos de la tupla Python nos devolverá un `TypeError` :

```
In [47]: colores[0] = "Café"
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-47-3502c7127536> in <module>()  
----> 1 colores[0] = "Café"  
  
TypeError: 'tuple' object does not support item assignment
```

1.4.8 Diccionarios

Los diccionarios son estructuras que contienen una colección de elementos de la forma `clave: valor` separados por comas y encerrados entre llaves. Las claves deben ser objetos inmutables y los valores pueden ser de cualquier tipo. Necesariamente las claves deben ser únicas en cada diccionario, no así los valores.

Vamos a definir un diccionario llamado `edades` en el cual cada clave será un nombre y el valor una edad:

```
In [49]: edades = {"Ana": 25, "David": 18, "Lucas": 35, "Ximena": 30, "Ale": 20}
```

Puede acceder a cada valor de un diccionario mediante su clave, por ejemplo, si quisieramos obtener la edad de la clave `Lucas` se tendría que escribir:

```
In [51]: edades["Lucas"]
```

```
Out[51]: 35
```

1.5 Operadores relacionales y lógicos

1.6 Control de flujo

1.6.1 Condicional if-elif-else

El condicional `if-elif-else` es una estructura de control que sirve para tomar decisiones en el flujo del programa. La sintaxis para `if-elif-else` es:

```

if cond1:
    # hacer algo
elif cond2:
    # hacer otra cosa
    .
    .
    .
elif condn:
    # hacer algo más
else:
    # hacer algo por default

```

Donde `cond1`, `cond2`, ... `condn` son valores lógicos que resultan de una comparación. Esta estructura se evalúa secuencialmente hasta encontrar una condición que se cumpla, si ninguna lo hace, entonces se ejecuta la instrucción colocada en el caso por default `else`.

1.6.2 Bucle for

El **bucle for** es una estructura de control de naturaleza repetitiva, en la cual se conocen *a priori* el número de iteraciones a realizar. En lenguajes como C++ o Java, el ciclo `for` necesita de una variable de ciclo de tipo entero que irá incrementándose en cada iteración. En Python, la cuestión es un poco diferente, el ciclo `for` *recorre* un *iterable* y en la k-ésima iteración la variable de ciclo *adopta* el valor del elemento en la k-ésima posición del iterable.

De manera general, la sintaxis de `for` es:

```

for var in iterable:
    # Hacer algo ...

```

Donde `var` es la **variable de ciclo** e `iterable` la secuencia de valores que deberá iterarse. Es necesario remarcar la importancia de los dos puntos al final de esta primera línea y en indentar el bloque de código subsecuente que definirá el cuerpo del ciclo `for`.

Como primer ejemplo vamos a recorrer una lista de números y mostrarlos por consola:

```

In [52]: numeros = [18,50,90,-20,100,80,37]
for n in numeros:
    print(n)

```

```

18
50
90
-20
100
80
37

```

Observe que en cada iteración la variable de ciclo `n` adopta el valor de cada uno de los elementos de la lista `numeros`.

Como ya se mencionó, en Python la variable de ciclo no necesariamente adopta valores numéricos enteros secuenciales, si no valores dentro de una secuencia. Esta secuencia podría ser también una cadena de caracteres, por ejemplo:

```

In [53]: palabra = "Python"
for letra in palabra:
    print(letra)

```

```

P
y
t
h
o
n

```

Dentro de un ciclo `for` podemos colocar cualesquiera otra instrucción de control de flujo. Un caso muy común es el de incluir otro ciclo `for`, algo que habitualmente se denota como **ciclos anidados**. Por ejemplo, supongamos que se requieren mostrar por

consola todos los elementos de algunas listas contenidas dentro de otra lista principal, en ese caso se hace necesario primero iterar sobre la lista principal y enseguida hacerlo sobre las listas contenidas, por ejemplo:

```
In [54]: matriz = [[-5,2,0], [9,5,6], [1,7,15]]
         for fila in matriz:
             for elemento in fila:
                 print(elemento)
```

```
-5
2
0
9
5
6
1
7
15
```

1.6.3 Bucle while

1.7 Funciones

Las funciones son *porciones de código* que nos sirven para modularizar nuestros programas y evitar en muchos casos la repetitividad de código. De manera general una función recibe algunos valores de entrada, los *procesa* y devuelve algunos valores de salida (o bien modifica algunas variables).

1.7.1 Funciones nativas de Python (built-in)

Python dispone de algunas funciones nativas que se *cargan* automáticamente cuando se inicia el intérprete. Por ejemplo la función `max` devuelve el mayor valor numérico de una lista de números:

```
In [55]: max([10,35,5,110,48,30,112,98,87])
```

```
Out[55]: 112
```

También existe una función `min`, análoga a `max`:

```
In [56]: min([10,35,5,110,48,30,112,98,87])
```

```
Out[56]: 5
```

Otro ejemplo de función nativa es `bin`, la cual dado un número en base 10 devuelve una cadena con la representación en base 2.

```
In [57]: bin(10)
```

```
Out[57]: '0b1010'
```

Naturalmente, el valor devuelto por una función se puede asignar a una variable y posteriormente ser utilizado:

```
In [66]: a = max([10,5,8])
         b = min([10,5,8])
         h = (a - b)/10
         print(h)
```

```
0.5
```

Hay funciones que no devuelven como tal un valor, si no que pueden modificar directamente alguna variable global o simplemente mostrar algo en la salida estándar como el caso de `print`.

Tendremos también funciones que aceptan más de un argumento, por ejemplo a la función `round` podemos pasarle dos argumentos: un número real y la cantidad de lugares decimales a considerar para el redondeo.


```
In [67]: round(3.141592653589793, 6)
```

```
Out[67]: 3.141593
```

```
In [68]: round(3.141592653589793, 2)
```

```
Out[68]: 3.14
```

1.7.2 Funciones definidas por el usuario

Además de las funciones nativas de Python, es posible definir nuestras propias funciones. En Python, de manera general, una función se define siguiendo la estructura mostrada a continuación:

```
def nombre_fun(arg1, arg2, ..., argN):  
    # Cuerpo de la función  
    # .  
    # .  
    # .  
    return val1, val2, ..., valN
```

Donde `def` es una palabra que debe anteceder siempre a la definición de una función, `nombre_fun` es el nombre que se asignará a la función, entre paréntesis y separados por comas se colocan los nombres de los argumentos de entrada, los dos puntos se colocan después de cerrar el paréntesis e indican que ahí termina el *encabezado* de la función y comenzará el *cuerpo* de la misma, aquí se colocarán todas las instrucciones que deberán realizarse; la palabra reservada `return` sirve para indicar los valores a devolver, mismos que se colocarán separados por comas.

Vamos a definir una función llamada `saluda`, la cual recibe un nombre (string) y devuelve un saludo (string) formado mediante concatenación:

```
In [70]: def saluda(nombre):  
         s = "Hola " + nombre + ", bienvenido."  
         return s
```

```
In [71]: print(saluda("Jorge"))
```

```
Hola Jorge, bienvenido.
```

Lo único que hace la función anterior es tomar un *string* como argumento y unirlo a algunas cadenas ya establecidas dentro de la función.

Veamos ahora cómo definir una función que recibe como argumento un entero y devuelve un valor lógico que indica si este es par.

```
In [73]: def espar(n):  
         if n%2 == 0:  
             s = True  
         else:  
             s = False  
         return s
```

```
In [75]: print(espar(2))  
         print(espar(5))  
         print(espar(10))
```

```
True  
False  
True
```

Naturalmente, las funciones pueden recibir más de un argumento. Por ejemplo:

```
In [76]: def mayor(a,b):  
        m = a  
        if a < b:  
            m = b  
        return m
```

```
In [77]: print( mayor(50,30) )  
print( mayor(1100,3050) )
```

```
50  
3050
```

La función `mayor` recibe dos valores numéricos y determina cuál es el mayor de ambos mediante una comparación con la sentencia `if`.

¿Pueden las funciones en Python devolver más de un valor? ¡Claro! Hace falta nada más separar con comas los valores a devolver.

```
In [79]: def calcula_rectangulo(b,h):  
        A = b*h  
        P = 2*b + 2*h  
        return A, P
```

```
In [80]: print( calcula_rectangulo(10,5) )  
print( calcula_rectangulo(50,15) )
```

```
(50, 30)  
(750, 130)
```

También es posible guardar/asignar los valores devueltos por la función en variables:

```
In [81]: area1, perimetro1 = calcula_rectangulo(100, 20)  
print("Área: {0}\nPerímetro: {1}".format(area1, perimetro1))
```

```
Área: 2000  
Perímetro: 240
```

1.7.3 Funciones con una cantidad de parámetros indeterminada

En ocasiones el número de parámetros que deberá recibir una función no puede ser algo fijo. Las definiciones de función en Python tienen la flexibilidad de poder recibir una cantidad variable de argumentos de entrada.

Para ejemplificar esto, vamos a crear una función llamada `promedio` que calcule el promedio de una cierta cantidad de números pasados como argumentos:

```
In [83]: def promedio(*numeros):  
        suma = 0  
        k = 0  
        for n in numeros:  
            suma += n  
            k += 1  
        return suma/k
```

```
In [84]: print(promedio(10,5))  
print(promedio(10,50,40,80,20,100))  
print(promedio(5,15,10,5))
```

```
7.5  
50.0  
8.75
```

Observe que lo único que hacemos es que al nombre del parámetro le anteponeamos un asterisco, esto le indica a Python que la cantidad de argumentos de entrada es indeterminada, en principio. Claro está, que el manejo posterior de la información es algo que el programador debe tener en cuenta. Dentro del cuerpo de la función se debe considerar que el parámetro `numeros` será

una tupla cuya cantidad de elementos dependerá de la cantidad de argumentos ingresados.

1.7.4 Funciones y los argumentos con nombre

Una función en Python se puede *mandar a llamar* pasándo los argumentos de manera posicional, es decir, en el orden que fueron definidos en la función, o bien, haciendo uso del nombre del parámetro correspondiente al argumento que se introduce, por ejemplo:

```
In [86]: def cuenta_cuantas(frase, letra):
        k = 0
        for car in frase:
            if car is letra:
                k += 1
        return k
```

```
In [87]: print( cuenta_cuantas("hola mundo", "o") )
        print( cuenta_cuantas(frase="hola mundo", letra="o") )
        print( cuenta_cuantas(letra="o", frase="hola mundo") )
```

```
2
2
2
```

La función `cuenta_cuantas` devuelve el número de presencias de una determinada letra en una frase. Observe las tres formas en que la *ejecutamos*, todas son equivalentes. En la primera se pasan los argumentos de forma posicional, en la segunda y tercera se utilizan los argumentos con nombres, note que en este caso el orden en que los argumentos son pasados, es indistinto.

En la definición de funciones es posible también especificar que se pasarán ciertos argumentos con nombre sin necesidad de escribirlos de manera explícita. Observe la siguiente función:

```
In [89]: def muestra_puntos(**personas):
        for persona in personas.items():
            print(persona[0] + " tiene " + str(persona[1]) + " puntos")
```

```
In [90]: muestra_puntos(Jorge=8, Paty=10)
        print(30*"=")
        muestra_puntos(Ana=6, Carlos=9, Victor=4, Daniela=8)
```

```
Jorge tiene 8 puntos
Paty tiene 10 puntos
=====
Ana tiene 6 puntos
Carlos tiene 9 puntos
Victor tiene 4 puntos
Daniela tiene 8 puntos
```

Vea que la definición de la función `muestra_puntos` incluye un parámetro llamado `**personas`, esos dos asteriscos antes del nombre del parámetro, indican que no se tiene predeterminado el número de argumentos que se pasarán, pero además, indica que cada argumento a introducir deberá ser un argumento con nombre. Dentro del cuerpo de la función el parámetro `**personas` es un diccionario cuyas claves son los nombres de los argumentos y los valores corresponden a cada valor asignado al argumento.

1.8 Ejercicios

1. En las siguientes opciones se muestran operaciones aritméticas entre diversos objetos de Python. Verifique si es posible realizarlas e indique el resultado, de no ser así describa el por qué.

- A. `1 + 2`
- B. `1.3 + 2.5`

- C. "1" + 2
- D. "1" + "2"
- E. [1,2,3] + [10,20]
- F. {"a":10, "b":5} + {"h":2, "i":4}

2. Observe el siguiente código e identifique y explique el error:

```
for a,b in [1,2,5,3,8,7]:  
    print(a)
```

3. El siguiente código debería imprimir la longitud de cada palabra contenida en la lista `palabras`. Identifique el error.

```
palabras = ["Carro", "Sol", "Mesa", "Dinosaurio", "Girasol", "Silla"]  
for palabra in palabras:  
    print(len(palabras))
```

- 4. Implemente un programa que determine si un número dado es par o impar.
- 5. Escriba un programa que cuente el número de vocales en una frase. Tome en cuenta que las vocales podrían ser tanto mayúsculas como minúsculas.
- 6. Escriba un programa que aproxime, mediante la suma de Riemann, el área bajo la curva de la función $f(x) = x^2 + 3x$ en el intervalo $0 \leq x \leq 10$.

In []: