El lenguaje de programación python

Índice

- Introduccción
- Fundamentos
- Control de flujo
- Estructuras de datos
- Modularidad
- Conclusiones

1. INTRODUCCIÓN

1. Introducción
Visión general

- Python es un lenguaje de programación interpretado y multiplataforma de propósito general:
  - Fácil de aprender, y aún así muy potente.
  - Estructuras de datos eficientes de alto nivel
  - Orientación a objetos sencilla pero efectiva
  - Sintaxis elegante
- Estas características lo convierten en uno de los más utilizados.

1. Introducción
Visión general

- La version 2 del lenguaje está discontinuada
  - Aquí utilizaremos la version 3
- Python puede utilizarse de diferentes maneras:
  - Desde la línea de comandos del intérprete de python.
  - Dentro de un IDE tal como Spyder o PyCharm (ficheros .py)
  - Como celdas de Código Fuente en un cuaderno interactivo Jupiter (ficheros .ipynb, Google Colab)

2. FUNDAMENTOS

2. Fundamentos
Modelo de datos

- Todos los datos en un programa Python estan representados por objetos
  - Incluso los números enteros!
- Cada objeto tiene una identidad, un tipo y un valor
  - La identidad de un objeto nunca cambia una vez creado; es la dirección en memoria de ese objeto
  - El operador “is” compara la identidad de dos objetos.
  - La función id() devuelve un entero representando su identidad.
  - La función type() devuelve el tipo del objeto.

2. Fundamentos
Modelo de datos

- Hay muchos tipos numéricos incluídos. Son inmutables; una vez creados su valor nunca cambia:
  - Enteros (int). Representan un rango ilimitado de números.
  - Booleanos (bool). Representan los valores Booleanos True y False.
  - Números en punto flotante (float). Representan números en punto flotante de doble precision a nivel de máquina.
- Los objetos de secuencias de tipos inmutables no pueden cambiarse unavez creados:
  - Cadenas de caracteres (str). Una cadena de caracteres es una secuencia de valores que representan caracteres Unicode.
  - Tuplas (tuple). Los elementos de una tupla son objetos Python arbitrarios.
- Los objetos de secuencias de tipos mutables pueden cambiarse una vez creados:
  - Listas (list). Los elementos de una lista son objetos Python arbitrarios.

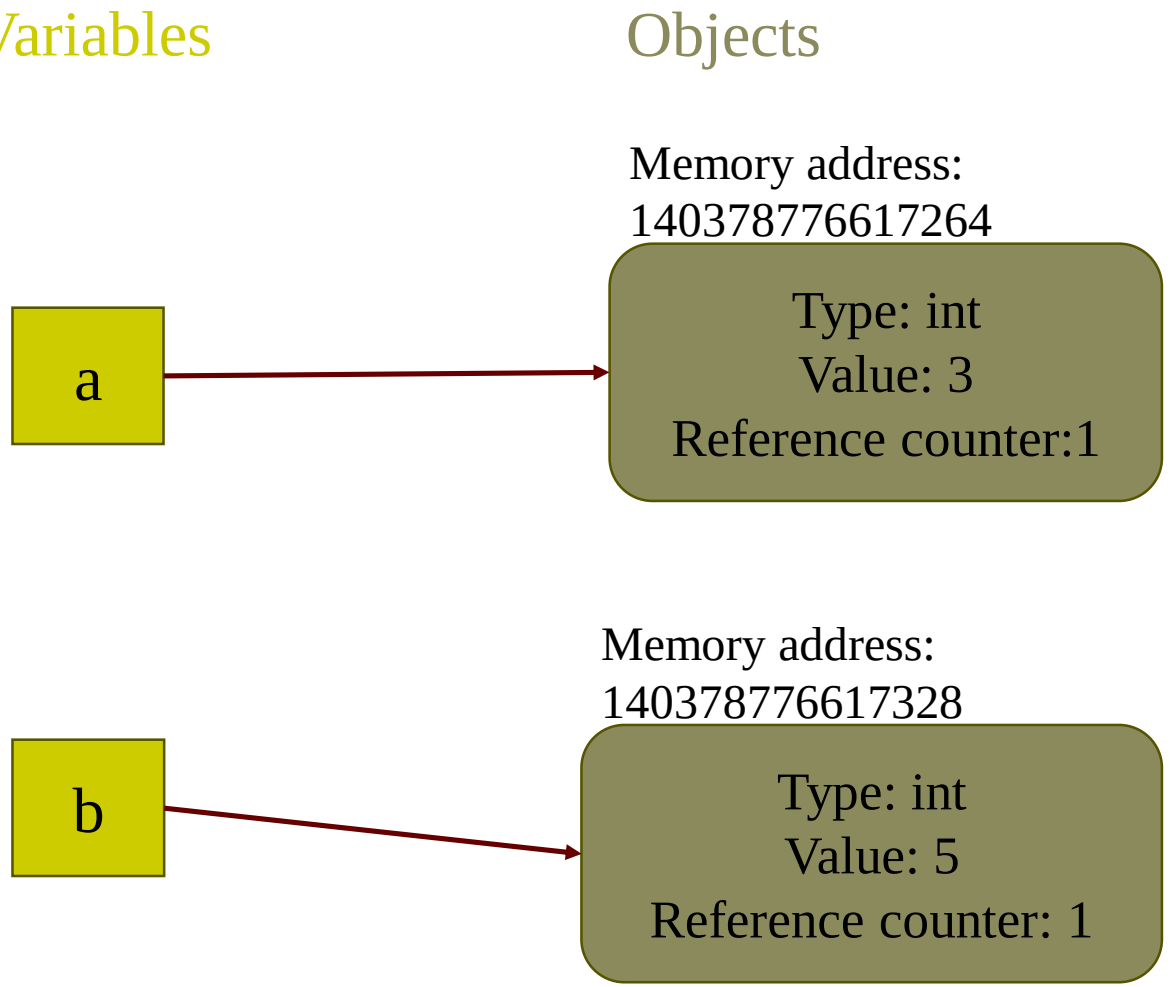
2. Fundamentos
Modelo de datos

- Las variables son referencias a objetos.
  - Cada variable contiene el nombre de la variable y un puntero al objeto.
- Cada objeto contiene un contador de referencias que cuenta el número de variables que apuntan a él.
  - Cuando el contador llega a cero, el objeto se marca para deslocalización por el gestor de memoria.
- La asignación de variables copia la referencia, no el objeto.
- El objeto nulo (None) puede ser asignado a cualquier variable.
- Hacer una copia del objeto puede tener sentido para objetos mutables:
  - La función copy() puede ser empleada para copiar listas.

2. Fundamentos

Modelo de datos

```
>>> a = 5
>>> print(a)
5
>>> print(id(a))
140378776617328
>>> b = a
>>> a = 3
>>> print(a)
3
>>> print(id(a))
140378776617264
>>> print(b)
5
>>> print(id(b))
140378776617328
```



2. Fundamentos
Tipado

- Las variables no tienen un tipo estático (tipado dinámico)
  - Una variable tiene el tipo del objeto al que está apuntando.
  - El tipo de una variable puede cambiar varias veces en tiempo de ejecución
  - Opcionalmente, pueden realizarse declaraciones de tipos pero suelen dar lugar a warnings en tiempo de compilación o ejecución.
- No obstante, no se pueden operar variables de tipos distintos a no ser que se realice una conversión de tipos explícita (tipado fuerte).

```
>>> print(5+"a")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

2. Fundamentos
Booleanos y enteros

- La clase bool solo tiene dos posibles valores: True y False
- Operadores booleanos: and, or, not
  - La evaluación en cortocircuito será empleada en and y or.
- La clase int permite valores arbitrariamente largos.
- Operadores aritméticos para enteros: +, -, *, //, %, **
- Operadores de asignación: =, +=, -=, *=, //=, %=, **=
- Operadores de comparación: ==, !=, >, <, >=, <=

2. Fundamentos
Números en punto flotante

- La clase float representa números en punto flotante de doble precision.
  - El operador aritmético / y el operador de asignación /= corresponden a la división en punto flotante.
- Las conversiones implícitas (promociones) se llevan a cabo para expresiones que mezclan variables de los tipos bool, int y float.
  - False se convierte a 0 y True se convierte a 1
- Las conversions explícitas pueden llevarse a cabo con bool(), int() y float()

2. Fundamentos
Cadenas de caracteres

- La clase str representa secuencias de valores que representan caracteres Unicode.
- Las cadenas constantes van entre comillas simples o dobles
  - Se pueden utilizer comillas simples para encerrar cadenas con comillas dobles y viceversa.
  - Las constantes de cadenas de caracteres multilínea se pueden especificar con comillas triples.
- El símbolo "\" se emplea para escapar caracteres especiales.
  - Nueva línea \n, tab \t
- Convertir a cadena de caracteres: str()
- Longitud de la cadena de caracteres: len()
- Leer cadena por teclado: input()
- Concatenación de cadenas de caracteres: +

2. Fundamentos
Troceado e indexado en Strings

- Se puede acceder individualmente a los caracteres de una string utilizando dos tipos de indexación:
  - Índices basados en cero (el índice 0 corresponde al primer caracter)
  - Índices negativos que comienzan desde el fin de la cadena hacia atrás (índice -1 corresponde al ultimo carácter)
- Las subcadenas se pueden extraer mediante troceado:
  - Copiar toda la cadena, [:]
  - Desde el principio hasta el fin de la cadena, [start:]
  - Desde el principio de la cadena hasta el final menos uno, [:end]
  - Desde el principio hasta el final menos uno, [start:end]
  - Desde el principio hasta el final menos uno en incrementos de tamaño “step”, [start:end:step]
  - Nótese que principio y/o fin pueden ser positivos negativos o cero
  - Nótese que el incremento puede ser positivo o negativo, pero no cero.

2. Fundamentos

Operaciones con cadenas de caracteres

- Comprobar subcadenas: substring in string, substring not in string
- Trocear cadena por separador: string.split(',')
- Eliminar espacios en blanco: string.strip(), string.lstrip(), string.rstrip()
- Encontrar la primera ocurrencia de una subcadena: string.find(substring)
- Encontrar el índice de la primera ocurrencia: string.index(substring)
- Contar ocurrencias: string.count(substring)
- Repetir n veces: string * n
- Reemplazar subcadena: string.replace(old, new, num_replacements)
- Interpolación de cadenas (f-strings):

```
>>> my_name = "John Smith"
>>> print(f"My name is: {my_name}\n")
My name is John Smith
```

3. CONTROL DE FLUJO

3. Control de flujo
Bloques de código y comentarios

- Los bloques de código no se marcan con delimitadores sino con el sangrado
- El número de espacios en blanco que marcan un bloque de código es normalmente de 2 o 4.
  - El criterio tiene que ser consistente a lo largo de todo un archivo de código fuente.
- Los comentarios se marcan con “#”
  - Se extienden hasta el fin de la línea
- Los docstrings se definen usando comillas triples.
  - Deberían estar colocados después de la definición de una clase o función
  - Deben ser descriptivos
  - Pueden consultarse con help(function) o help(class)

3. Control de flujo
La sentencia if

```
if condition:
    code_block

if condition:
    code_block1
else:
    code_block2

if condition1:
    code_block1
elif condition2:    # More elif clauses can follow
    code_block2
else:
    code_block3
```

3. Control de flujo
La sentencia match-case

```
match expression:
    case pattern_1:
        code_block_1
    case pattern_2:
        code_block_2
…
    case pattern_n:
        code_block_n
    case _:     # If the expression does not match any of the previous patterns
        code_block_final
```

3. Control de flujo
La sentencia While

- Estructura básica del bucle:

```
while condition:
    code_block
```

- La sentencia continue puede ser utilizada para obviar el resto del bloque de código y evaluar la condición de nuevo.
- La sentencia break se puede utilizar dentro del bloque de Código para terminar el bucle de manera inmediata sin esperar a que se cumpla la condición.
- Se puede comprobar si el bucle terminó normalmente o mediante una sentencia break:

```
while condition:
    code_block
else:
    code_block_else # Executes if the loop ended by execution of break
```

3. Control de flujo
La sentencia for

- Itera a través de un objeto iterable: string, list, dictionary, file, etc.
- Estructura del bucle:

```
for element in iterable_object:
    code_block
```

- Las sentencias break y continue pueden ser utilizadas dentro de este bloque de código.
  - La cláusula else puede añadirse al final para comprobar si el bucle terminó por una sentrencia break
- El objeto iterable más frecuente es una secuencia de enteros
  - Se utiliza la función range(start, end, step)
  - El parámetro start es opcional, por defecto 0
  - El parámetro end es obligatorio, la secuencia alcanza end menos uno.
  - El parámetro step es opcional, por defecto 1

4. ESTRUCTURAS DE DATOS

4. Estructuras de datos

Listas

- Las listas son mutables, es decir, pueden cambiar después de su creación.
- Guardan objetos de cualquier tipo con un orden definido, se permiten objetos duplicados
- Las listas se crean especificandola secuencia de objetos separados por comas y encerradas por corchetes:

```
>>> empty_list = []
>>> my_list = ["Python", 43, True, 43, -3.2836]
```

- La función list() se puede utilizer para convertir otros tipos de datos en listas:

```
>>> list("Python")
['P', 'y', 't', 'h', 'o', 'n']
>>> list(range(5))
[0, 1, 2, 3, 4]
```

4. Estructuras de datos
Indexado y troceado de listas

- La sintaxis es la misma que la de cadenas de caracteres:

```
>>> my_list[-2]
43
>>> my_list[1:5:2]
[43, 43]
```

- Indexar un elemento fuera de rango lanzará un error
- Los elementos fuera de rango serán ignorados en el troceado. Es decir, no se lanza error.
- Longitud de una lista: len(my_list)
- Se puede hacer una copia de la lista con my_list[:]
- Se puede hacer una copia invertida de la lista con my_list[::-1]

4. Estructuras de datos

Operaciones con listas

- Copiar la lista: my_list.copy(), my_list.deepcopy()
- Invertir la lista guardando la original: reversed(my_list)
- Invertir la lista modificando la original: my_list.reverse()
- Añadir al final: my_list.append(element)
- Insertar en posicion arbitraria: my_list.insert(pos, element)
- Repetir n veces: my_list * n
- Concatenar guardando la original: +, +=
- Concatenar modificando la original: my_list.extend(list2)
- Modificar/Eliminar una sublista: my_list[start:end] = list2
  - list2 puede tener una longitude diferente a la de la sublista sustituida.
  - If list2==[], la sublista es borrada

4. Estructuras de datos

Operaciones con listas

- Borrar un elemento por su índice: del my_list[index]
- Borrar un elemento por su valor (primera ocurrencia): my_list.remove(element)
- Borrar un elemento por su índice, con extracción: element = my_list.pop(index)
- Vaciar la lista, es decir borrar todos sus elementos: my_list.clear()
  - Esto es diferente de my_list = [], que crea una lista nueva vacía.
- Comprobar si un elemento existe: element in my_list
- Número de ocurrencias: my_list.count(element)
- Ordenar conservando el original: sorted(my_list)
- Ordenar modificando el original: my_list.sort()

4. Estructuras de datos

Iterar en listas

- Versión básica:

```
for element in my_list:
    code_block
```

- Enumerar los elementos comenzando por 0:

```
for index, element in enumerate(my_list):
    code_block
```

- Tomar un elemento de cada lista hasta que la lista más corta se acabe:

```
for element1, element2 in zip(list1, list2):
    code_block
```

4. Estructuras de datos
Listas por comprensión

- Versión básica:

```
my_list = [value for item in iterable]
>>> [character.upper() for character in "Python"]
['P', 'Y', 'T', 'H', 'O', ‘N’]
```

- Versión con condición:

```
my_list = [value for item in iterable if condition]
>>> [c.upper() for c in "Python" if c>’m’]
['Y', 'T', 'O', 'N’]
```

- Versión anidada:

```
>>> [c1.upper()+c2.lower() for c1 in "Py" for c2 in "th"]
['Pt', 'Ph', 'Yt', 'Yh']
```

4. Estructuras de datos

Tuplas

- Las tuplas son muy similares a las listas, solo que las tuplas son inmutables, es decir, no pueden ser modificadas después de ser creadas.
  - Intentar modificar una tupla lanzará un error en tiempo de ejecución.
- Las tuplas se crean especificando la secuencia de objetos separados por comas y encerrados entre paréntesis.

```
>>> empty_tuple = ()
>>> one_element_tuple = ("This element", )   # La coma es obligatoria
>>> other_tuple = (-5.2783, "Hello", True, 826)
```

- Dado que hay al menos una coma, los paréntesis se pueden omitir.

```
>>> one_element_tuple = "This element",    # La coma es obligatoria
>>> other_tuple = -5.2783, "Hello", True, 826
```

4. Estructuras de datos

Tuplas

- Otros tipos de datos iterables pueden ser convertidos en tuplas con la función tuple():

```
>>> tuple(["One", "Two", "Three"])
("One", "Two", "Three")
```

- Todas las operaciones definidas para listas también funcionan para tuplas, excepto aquellas que modifican la lista original.
- Desempaquetado de tuplas en varias variables:

```
>>> a, b, c = ("One", "Two", "Three")
```

- Intercambio de variables utilizando desempaquetado de tuplas:

```
>>> a, b = b, a
```

4. Estructuras de datos

Tuplas

- El desempaquetado de tuplas extendido permite el agrupamiento de de varios objetos de la tuple desempaquetada:

```
>>> my_tuple = ('G', 'A', 'R', 'Y', 'W')
>>> head, *body, tail = my_tuple
>>> head
'G'
>>> body
['A', 'R', 'Y']
>>> tail
'W'
```

- El desempaquetado también puede aplicarse a otros tipos de datos iterables.

4. Estructuras de datos

Diccionarios

- Los diccionarios (dicts) tienen las siguientes características:
  - Almacenan valores indexados por claves.
  - Conservan el orden en el que las claves se insertan.
  - Son mutables
  - Las claves deben ser únicas. Cualquier tipo de datos inmutable puede constituir una clave.
  - La función hash() incluida se usa para aplicar hash a las claves.
  - El acceso a los elementos es muy rápido.
- Los diccionarios se crean a partir de secuencias clave:valor que son rodeadas por llaves:

```
>>> empty_dict = {}
>>> other_dict = { "a": 572, "b": 826, "c": -4844, "d": 572}
```

4. Estructuras de datos
Diccionarios

- Se accede a los elementos especificando la clave entre corchetes para leer, modificar o insertar dichos elementos:

```
>>> other_dict["d"]
572
```

- Intentar acceder a una clave inexistente dará un error en tiempo de ejecución.
  - Comprobar si la clave existe: key in my_dictionary, key not in my_dictionary
  - La función get() devuelve el valor si la clave existe y None si la clave no existe.
- Número de pares clave-valor: len(my_dictionary)

4. Estructuras de datos

Diccionarios

- Obtener todas las claves: my_dictionary.keys()
- Obtener todos los valores: my_dictionary.values()
- Obtener todos pares clave-valor como una lista de tuplas: my_dictionary.items()
- Para iterar sobre un diccionario bastan las tres funciones anteriores.
- Dos o más diccionarios pueden ser mezclados en uno
  - En caso de colisión de claves se conserva el valor del último diccionario.
- Mezcla conservando los diccionarios originales:

```
>>> {**dict1, **dict2}
>>> dict1 | dict2
```

- Mezcla modificando uno de los diccionarios:

```
>>> dict1.update(dict2)
```

4. Estructuras de datos

Diccionarios

- Borrado por clave:

```
>>> del my_dict[key]
```

- Borrado con extracción:

```
>>> element = my_dict.pop(key)
```

- Vaciar el diccionario completamente (borrar todos los pares clave-valor): my_dict.clear()
  - Esto es diferente de my_dict = {}, que crea un nuevo diccionario vacío.
- Diccionarios por comprensión:

```
>>> my_dict = {w: len(w) for w in words if w[0] not in "aeiou"}
```

4. Estructuras de datos

Conjuntos

- Un conjunto contiene varios valores únicos sin un orden específico.
- Los conjuntos son mutables
- Los elementos deben ser hashable o “encriptables”
- Un conjunto puede crearse separando los valores por comas y rodeándolos por llaves:

```
>>> numbers = { 67, -34, 82, -111}
```

- El conjunto vacío se crea con la función set() sin argumentos
- Otros tipos de datos iterables pueden ser convertidos a conjuntos con la función set()
  - Los valores duplicados serán borrados antes de la creación del conjunto

4. Estructuras de datos

Conjuntos

- Añadir elemento: my_set.add(element)
- Borrar elemento: my_set.remove(element)
- Número de elementos (cardinal): len(my_set)
- Comprobar si un elemento existe: element in my_set, element not in my_set
- Congelar un conjunto (hacerlo inmutable): frozenset(my_set)
- Iterar sobre un conjunto: for element in my_set
- Operaciones con conjuntos: intersection (&), union (|), difference (-), symmetric difference (^), inclusion (<, <=, >, >=)
- Conjuntos por comprensión:

```
>>> my_set = {n for n in range(0, 20) if n % 3 == 0}
```

5. MODULARIDAD

5. Modularidad

Funciones

- Definición de funciones con su lista de parámetros:

```
def function_name(parameter1, parameter2):
    function_body
```

- El cuerpo de la función debe contener al menos una sentencia.
  - La sentencia pass es una sentencia de no operación.
- La orden return devolverá un valor. En caso contrario devolverá None
  - Si se utiliza return sin devolver ningún valor, se devolverá None.
- Se pueden devolver múltiples valores como una tupla.
- Llamada a una función con su lista de argumentos:

```
function_name(argument1, argument2)
```

5. Modularidad

Funciones

- Los argumentos posicionales son aquellos que son asignados a sus correspondientes parámetros por orden
- Los argumentos nominales son copiados asignando cada parámetro por su nombre:

function_name(parameter2 = argument1, parameter1 = argument2)

- Los argumentos posicionales y nominales pueden mezclarse, pero los argumentos posicionales deben ir primero.
- Los valores por defecto para los parámetros pueden especificarse en la definición de la función.

def function_name(parameter1, parameter2 = 0.0):

5. Modularidad

Funciones

- El paso de parámetros se realiza por referencia
- Cada argumento se asigna a su parámetro correspondiente como en una asignación estándar de variable, es decir copiando la referencia del objeto.
- Para objetos inmutables esto significa que la modificación realizada en los parámetros dentro de la función no se podrá ver fuera de las mismas.
- Para objetos mutables esto significa que las modificaciones en los objetos referenciados por los parámetros se verán fuera de la propia función.
- Si asignamos un nuevo objeto a un parámetro dentro de una función, dicho cambio no se verá fuera de la misma.

5. Modularidad

Funciones

- Se debería añadir documentación a la definición de cada función insertando un docstring entre triples comillas al principio del cuerpo de dicha función:

```
def my_function(param1, param2):
    ''' This is the docstring of the function
    …
    '''
    rest_of_function_body
```

- La función help() se puede utilizar para mostrar el docstring de una función:

```
>>> help(my_function)
```

5. Modularidad

Funciones

- Las anotaciones de tipos o “type hints” pueden añadirse para indicar los tipos esperados para los parámetros y el valor de retorno

```
def my_function(param1: str, param2: int) -> dict[str, float]:
```

- Las variables también pueden ser anotadas con o sin asignación:

```
>>> age: int = 1
>>> year: int
```

- El intérprete de Python no impone anotaciones de tipo de funciones y variables
  - Pueden ser usadas por herramientas de terceros como comprobadores de tipos, IDEs, etc.

5. Modularidad
Programación orientada a objetos

- Los métodos de instancia tienen un primer parámetro self que referencia la instancia actual.
- Los métodos de clase tienen un primer parámetro cls que referencia la clase actual
  - El decorador @classmethod debe ser especificado antes de la definición del método
- El constructor es el método de instancia __init__
- Los atributos de instancia son accesibles a través de self.attribute_name desde dentro de los métodos de instancia o object_name.attribute_name desde fuera de los métodos de instancia
  - Se suelen definir dentro del constructor
- Los atributos de clase son accesibles a través de cls.attribute_name desde dentro de los métodos de la clase o class_name.attribute_name desde fuera de los métodos de la clase
  - Se suelen definir fuera del constructor

5. Modularidad
Programación orientada a objetos

```
class Droid:
    droid_counter = 0

    @classmethod
    def num_droids(cls) -> int:
        return cls.droid_counter

    def __init__(self):
        self.power_on = False
        Droid.droid_counter += 1

    def switch_on(self):
        self.power_on = True

    def switch_off(self):
        self.power_on = False
```

5. Modularidad

Módulos

- Un módulo es un fichero fuente con extension .py
- Los módulos pueden agruparse en carpetas llamadas paquetes
- Las carpetas pueden agruparse en librerías (o bibliotecas)
- Algunas veces los términos “paquete” y “librería” se usan indistintamente
- Los módulos pueden importarse completos:

```
>>> import stats
>>> stats.std(6, 3, 9, 5)
2.5
```

- También pueden importarse funciones específicas de un modulo:

```
>>> from stats import mean
>>> mean(6, 3, 9, 5)
5.75
```

5. Modularidad

Módulos

- Se pueden utilizar alias para módulos o funciones:

```
>>> from stats import mean as avg
>>> avg(6, 3, 9, 5)
5.75
>>> import stats as mystatistics
>>> mystatistics.std(6, 3, 9, 5)
2.5
```

- El módulo principal de un Proyecto se suele nombrar main.py
- Punto de entrada del programa principal:

```
if __name__ == '__main__':
    # Este es el punto de entrada del programa principal
```

- Cada módulo puede ser importado o ejecutado como un programa principal.

6. CONCLUSIÓN

6. Conclusión

- Algunas de las características del lenguaje deben ser utilizadas con cuidado para evitar "hard-to-spot" bugs:
  - Tipado dinámico, es decir el tipo de una variable puede cambiar en tiempo de ejecución
  - Las estructuras de datos pueden contener elementos de tipos de datos mezclados
  - Colisiones de nombres al importar librerías
- Consejos para aumentar el rendimiento:
  - Utilizar listas por comprensión antes que bucles
  - Emplear las funciones incluídas
  - Utilizar "in" siempre que sea posible
  - Utilizar conjuntos antes que bucles
  - Utilizar el desempaquetado de tuplas