Modulo 4 - Funciones, Tuplas, Diccionarios, Exceptiones y Procesamiento de Datos

- Estructuración de código y el concepto de función.
- Invocación de funciones y devolución de resultados de una función.
- Alcance de nombres y sombreado de variables.
- Tuplas y su propósito: construcción y uso de tuplas.
- Diccionarios y su propósito: construcción y uso de diccionarios.
- Introducción a las excepciones en Python.

¿Por qué necesitamos funciones?

Hasta ahorita has implementado varias veces el uso de funciones, pero solo se han visto algunas de sus ventajas. Solo se han invocado funciones para utilizarlas como herramientas, con el fin de hacer la vida más fácil, y para simplificar tareas tediosas y repetitivas.

Cuando se desea mostrar o imprimir algo en consola se utiliza print(). Cuando se desea leer el valor de una variable se emplea input(), combinados posiblemente con int() o float().

También se ha hecho uso de algunos métodos, los cuales también son funciones, pero declarados de una manera muy específica.

Ahora aprenderás a escribir tus propias funciones, y como utilizarlas. Escribiremos varias de ellas juntos, desde muy sencillas hasta algo complejas. Se requerirá de tu concentración y atención.

Muy a menudo ocurre que un cierto fragmento de código se repite muchas veces en un programa. Se repite de manera literal o, con algunas modificaciones menores, empleando algunas otras variables dentro del programa. También ocurre que un programador ha comenzado a copiar y pegar ciertas partes del código en más de una ocasión en el mismo programa.

Puede ser muy frustrante percatarse de repente que existe un error en el código copiado. El programador tendrá que escarbar bastante para encontrar todos los lugares en el código donde hay que corregir el error. Además, existe un gran riesgo de que las correcciones produzcan errores adicionales.

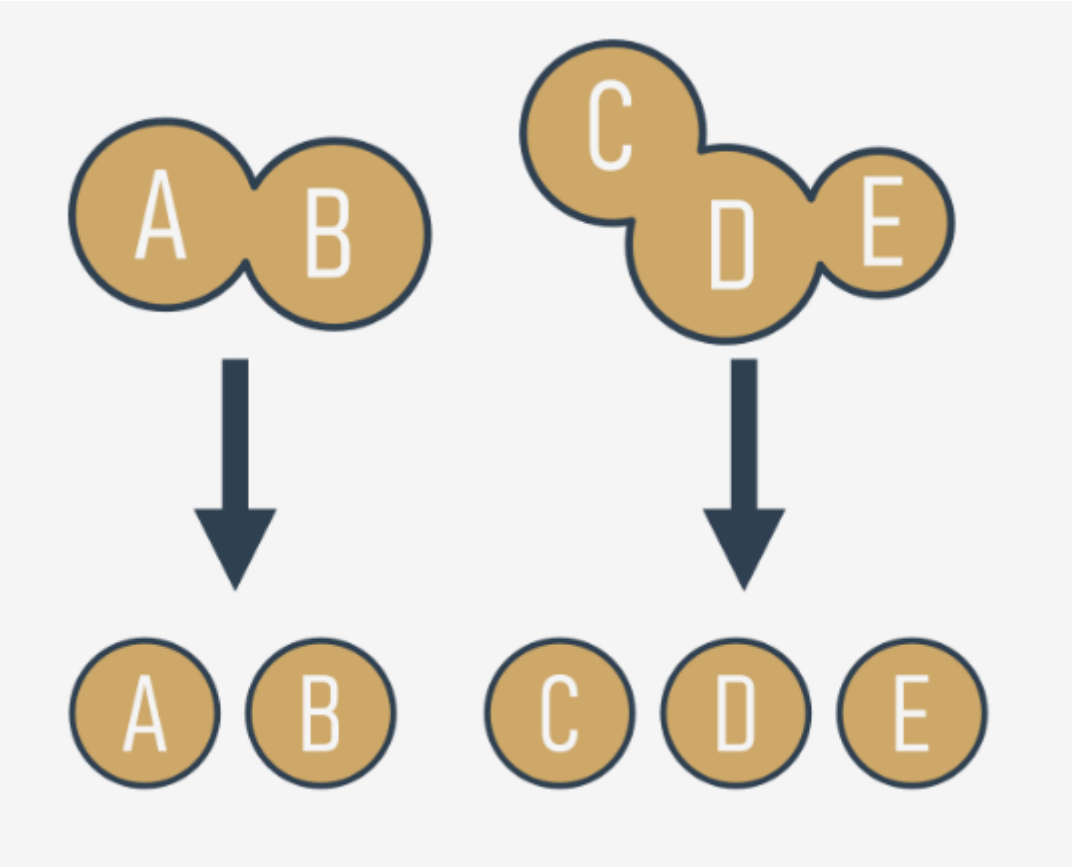
Definamos la primer condición por la cual es una buena idea comenzar a escribir funciones propias: si un fragmento de código comienza a aparecer en más de una ocasión, considera la posibilidad de aislarlo en la forma de una función invocando la función desde el lugar en el que originalmente se encontraba.

Puede suceder que el algoritmo que se desea implementar sea tan complejo que el código comience a crecer de manera incontrolada y, de repente, ya no se puede navegar por él tan fácilmente.

Se puede intentar solucionar este problema comentando el código, pero pronto te darás cuenta que esto empeorará la situación - demasiados comentarios hacen que el código sea más difícil de leer y entender. Algunos dicen que una función bien escrita debe ser comprensible con tan solo una mirada.

Un buen desarrollador divide el código (o mejor dicho: el problema) en piezas aisladas, y codifica cada una de ellas en la forma de una función.

Esto simplifica considerablemente el trabajo del programa, debido a que cada pieza se codifica por separado y consecuentemente se prueba por separado. A este proceso se le llama comúnmente descomposición.

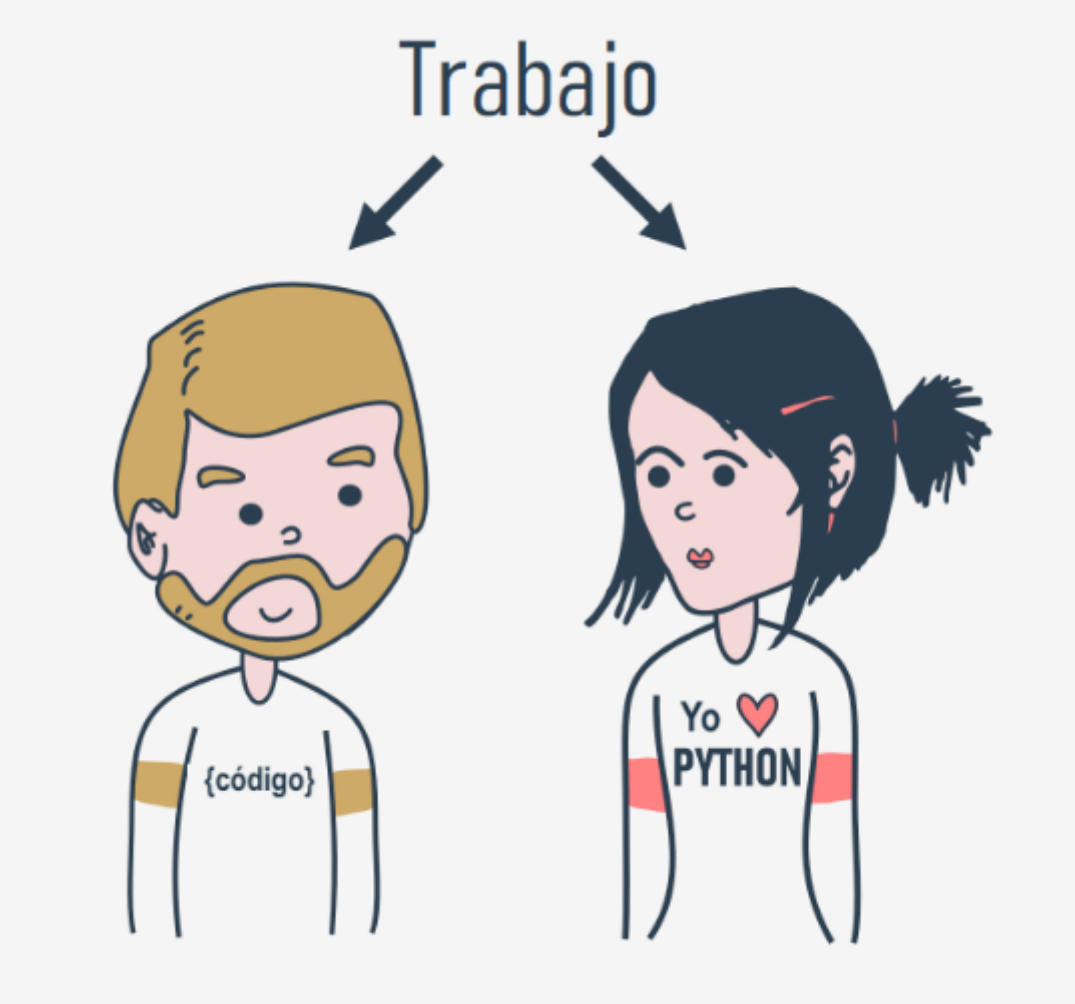


Existe una segunda condición: si un fragmento de código se hace tan extenso que leerlo o entenderlo se hace complicado, considera dividirlo pequeños problemas por separado e implementa cada uno de ellos como una función independiente.

Esta descomposición continúa hasta que se obtiene un conjunto de funciones cortas, fáciles de comprender y probar.

Descomposición

Es muy común que un programa sea tan largo y complejo que no puede ser asignado a un solo desarrollador, y en su lugar un equipo de desarrolladores trabajarán en el. El problema, debe ser dividido entre varios desarrolladores de una manera en que se pueda asegurar su eficiencia y cooperación.



Es inconcebible que más de un programador deba escribir el mismo código al mismo tiempo, por lo tanto, el trabajo debe de ser dividido entre todos los miembros del equipo.

Este tipo de descomposición tiene diferentes propósitos, no solo se trata de compartir el trabajo, sino también de compartir la responsabilidad entre varios desarrolladores.

Cada uno debe escribir un conjunto bien definido y claro de funciones, las cuales al ser combinadas dentro de un módulo (esto se clarificara un poco más adelante) nos dará como resultado el producto final.

Esto nos lleva directamente a la tercera condición: si se va a dividir el trabajo entre varios programadores, **se debe descomponer el problema para permitir que el producto sea implementado como un conjunto de funciones escritas por separado empacadas juntas en diferentes módulos**.

¿De dónde provienen las funciones?

En general, las funciones provienen de al menos tres lugares:

- De Python mismo: varias funciones (como print()) son una **parte integral de Python**, y siempre están disponibles sin algún esfuerzo adicional del programador; se les llama a estas **funciones integradas**.
- De los **módulos preinstalados** de Python: muchas de las funciones, las cuales comúnmente son menos utilizadas que las integradas, están disponibles en módulos instalados juntamente con Python; para poder utilizar estas funciones el programador debe realizar algunos pasos adicionales (se explicará acerca de esto en un momento).
- **Directamente del código**: tu puedes escribir tus propias funciones, colocarlas dentro del código, y usarlas libremente.
- Existe una posibilidad más, pero se relaciona con clases, se omitirá por ahora.

Tu primera función

Se necesita definirla. Aquí, la palabra definir es significativa.

Así es como se ve la definición más simple de una función:

```
def function_name():
    function_body
```

- Siempre comienza con la palabra reservada def (que significa definir).
- Después de def va el nombre de la función (las reglas para darle nombre a las funciones son las mismas que para las variables).
- Después del nombre de la función, hay un espacio para un par de paréntesis (ahorita no contienen algo, pero eso cambiará pronto).
- La línea debe de terminar con dos puntos.
- La línea inmediatamente después de def marca el comienzo del cuerpo de la función - donde varias o (al menos una) instrucción anidada será ejecutada cada vez que la función sea invocada; nota: la función termina donde el anidamiento termina, se debe ser cauteloso.

A continuación se definirá la función. Se llamará **message**:

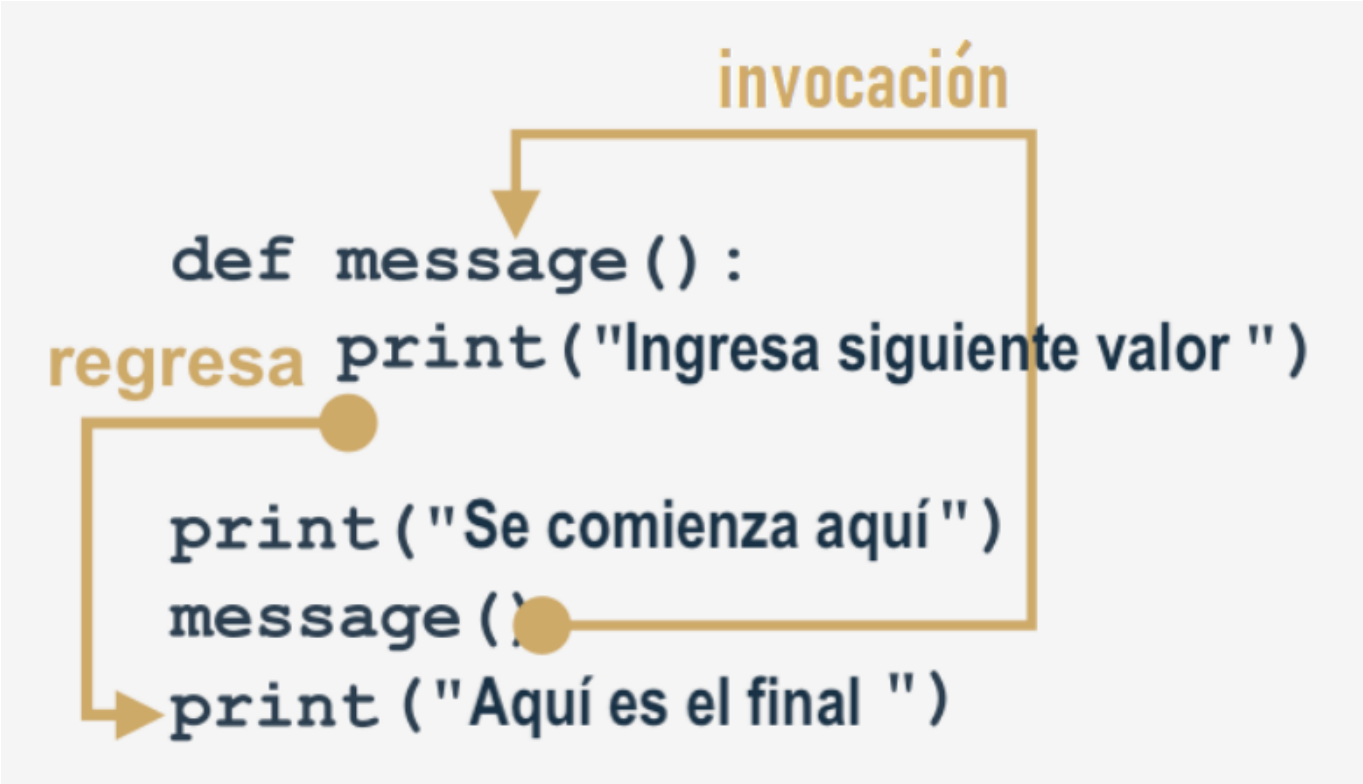
```
def message():
    print("Ingresa un valor: ")
```

Se ha modificado el código, se ha insertado la invocación de la función entre los dos mensajes:

```
def message():
    print("Ingresa un valor: ")

print("Se comienza aquí.")
message()
print("Se termina aquí.")
```

el funcionamiento de las funciones



Intenta mostrarte el proceso completo:

- Cuando se invoca una función, Python recuerda el lugar donde esto ocurre y salta hacia dentro de la función invocada.
- El cuerpo de la función es entonces ejecutado.
- Al llegar al final de la función, Python regresa al lugar inmediato después de donde ocurrió la invocación.

Existen dos consideraciones muy importantes, la primera de ella es:

No se debe invocar una función antes de que se haya definido.

Recuerda: Python lee el código de arriba hacia abajo. No va a adelantarse en el código para determinar si la función invocada está definida más adelante, el lugar correcto para definirla es antes de ser invocada.

Una función y una variable no pueden compartir el mismo nombre.

El asignar un valor al nombre «message» causa que Python olvide su rol anterior. La función con el nombre de message ya no estará disponible.

Afortunadamente, es posible combinar o mezclar el código con las funciones - no es forzoso colocar todas las funciones al inicio del archivo fuente.

Funciones parametrizadas

El potencial completo de una función se revela cuando puede ser equipada con una interface que es capaz de aceptar datos provenientes de la invocación. Dichos datos pueden modificar el comportamiento de la función, haciéndola más flexible y adaptable a condiciones cambiantes.

Un parámetro es una variable, pero existen dos factores que hacen a un parámetro diferente:

- **Los parámetros solo existen dentro de las funciones en donde han sido definidos**, y el único lugar donde un parámetro puede ser definido es entre los paréntesis después del nombre de la función, donde se encuentra la palabra reservada def.

- **La asignación de un valor a un parámetro de una función se hace en el momento en que la función se manda llamar o se invoca**, especificando el argumento correspondiente.

```
def function(parameter):
    ###
```

Recuerda que:

- Los parámetros solo existen dentro de las funciones (este es su entorno natural).
- Los argumentos existen fuera de las funciones, y son los que pasan los valores a los parámetros correspondientes.
- Especificar uno o más parámetros en la definición de la función es un requerimiento, y se debe de cumplir durante la invocación de la misma. Se debe proveer el mismo número de argumentos como haya parámetros definidos.

Existe una circunstancia importante que se debe mencionar.

Es posible tener una variable con el mismo nombre del parámetro de la función.

El siguiente código muestra un ejemplo de esto:

```
def message(number):
    print("Ingresa un número:", number)

number = 1234
message(1)
print(number)
```

Una situación como la anterior, activa un mecanismo denominado **sombreado**:

- El parámetro x sombrea cualquier variable con el mismo nombre, pero…
- … solo dentro de la función que define el parámetro.

El parámetro llamado number es una entidad completamente diferente de la variable llamada number.

Una función puede tener tantos parámetros como se desee, pero entre más parámetros, es más difícil memorizar su rol y propósito.

Paso de parámetros posicionales

La técnica que asigna cada argumento al parámetro correspondiente, es llamada **paso de parámetros posicionales**, los argumentos pasados de esta manera son llamados **argumentos posicionales**.

Paso de argumentos con palabra clave

From:
https://miguelangel.torresegea.es/wiki/ - **miguel angel torres egea**

Permanent link:
https://miguelangel.torresegea.es/wiki/info:cursos:netacad:python:pe1m4?rev=1655232044

Last update: **14/06/2022 11:40**