Python 3.5.2 tutorial: funciones

Hoy día continuamos el aprendizaje: veremos las funciones en Python ¡adelante!

Introducción.

Esta es la segunda parte de nuestro tutorial sobre Python, específicamente la versión 3.5.2, para ser exactos. Para mayores detalles de compatibilidad, historia y, en fin, lo básico sobre este lenguaje, debéis leer primero nuestra entrega anterior. Pero si ya tenéis alguna experiencia, podéis comenzar aquí con funciones. Y si soís un profesional en esto, **gracias por vuestra visita**, vuestras correcciones las podéis hacer llegar vía Twitter a @ks7000 o realizar un comentario acá abajo al finalizar esta entrada. Pues bien, *¡manos a la obra!*

Funciones en Python.

Clasificación de las funciones.

En la primera parte de nuestro tutorial hicimos una sencilla función, demasiado simple, para nuestro programa **que_hubo_mundo.py**, y ahora es bueno que veaí? el panorama completo de las funciones *desde nuestro punto de vista.*

Lo aquí expresado por nosotros es nuestra opinión, la forma y manera en que enfocamos el aprendizaje del lenguaje Pyhton. *En la página web oficial podréis ver el tutorial completo -en inglés- y con todo detalle.* Nuestra idea es que aprendáis lo básico y una vez estéis diestro o diestra en la materia, repaséis en Python.org y contrásteis la información. ¿Queréis continuar? **¡Vamos!**

Las funciones en Python las consideramos en el siguiente orden:

- Funciones nativas o integradas: no se necesita acción alguna fuera de llamarlas y pasarles los argumentos, si es que exige alguno. Ejemplo fácil: la función **print()** la cual podemos llamarla y simplemente nos "devuelve" un retorno de carro por cónsola en pantalla. Y decimos que nos "devuelve" porque eso es lo que vemos que hace la función, porque en realidad no nos devuelve valor alguno, ni numérico ni de texto. ¿Recordáis la palabra clave "**end**" para **print()**? Ingresad en vuestra cónsola lo siguiente (le "pasamos" un argumento a

print() *especificamente un argumento con palabra clave*): **print(end="")** *¿Qué os parece?*

- Funciones que vienen con Python pero hay que declararlas primero para luego llamarlas: pues eso, vamos a traer un ejemplo práctico, cuando vosotros estáis trabajando en modo interactivo tal vez os gustaría poder "limpiar" la cónsola de todo lo que hayáis practicado. Pues bien, esta tarea descansa con el sistema operativo de vuestro ordenador, y en el caso de GNU/Linux se hace con el comando "**clear**" pero antes que intentéis probarla revisad primero el siguiente código, observad la palabra clave o comando **import**:
- >>> import os
  >>> os.system("clear")
  0
  >>>
- Eso es lo que veréis, más o menos. La pantalla se "limpiará" pero obervareís que aparece un cero y luego el prompt de Python interactivo. *Por eso es que es una función, devuelve un valor, que en este caso es cero.* Si queremos que ese cero no aparezca y veamos una prístina cónsola (solo el prompt, por su puesto) deberemos teclear algo como esto: **k = os.system("clear")**. Al finalizar estos párrafos os colocamos una captura de pantalla para que veaís más detalles.
- Funciones para Python que hay que descargarlas e importarlas: nosotros podemos publicar (como de hecho lo hicimos) nuestro archivo **que_hubo_mundo.py** el cual contiene la función **hola()** la cual solamente imprime el mensaje "¡Qué hubo mundo!". Abrid una ventana terminal, listad los ficheros con el comando ls y si tenéis el archivo en cuestión en esa carpeta entonces abrid la terminal interactiva e ingresad lo siguiente: **from que_hubo_mundo import hola** y observad lo que pasa. ¡Genial! ¿o no? Pues bueno así podéis crear tus "librerías" e incluir el trabajo de otros, *así, sin mayor parsimonia ni protocolo (punto 3 PEP20).* El problema de esto es que si traemos el trabajo de otras personas hasta nuestro disco duro y dichos programadores publican una nueva versión (para mejor, se supone) ***deberemos descargar manualmente y verificar si funciona con nuestro programa -dos trabajos en uno-***. Para evitar esto pasemos al siguiente punto.
- Funciones para Python que hay que instalarlos por un procedimiento especial: Python.org mantiene siempre sus puertas abiertas a contribuciones, tanto monetarias como de programas y para ello mantiene en línea "the Python Package Index" (El Índice de Paquetes de Python) con el cual podemos instalar el trabajo de los demás y mantenernos actualizados si surgen nuevas versiones. El "procedimiento especial" consiste en instalar el **programa pip** el cual será el que haga el trabajo por nosotros (*aunque probablemente ya lo tenéis instalado*). A su vez, para instalar pip debemos descargar un archivo llamado **get-pip.py** desde un servidor seguro (HTTPS, "candado verde") y ejecutarlo. Luego podremos usar directamente desde Python.org cualquier trabajo publicado. Aquí es importante acotar que se deben seguir normas y reglas para publicar y son de obligatorio cumplimiento si queremos llegar lejos en esto. Sabemos que si seguimos estudiando y trabajando duro algún día publicaremos algo allí pero **si queréis probar** cómo funciona (declarar, subir y luego bajar nuestra programación) Python ofrece un ambiente de prueba (le dicen

"sandbox" en inglés ya que allá se acostumbra poner en los parques cajones de madera llenos de arena para que los niños y niñas desarrollen sus habilidades motoras sin dañar nada ni herirse): he aquí el enlace y "happy hacking!" como dicen allá ;-) .

Creación de una función con el comando pass.

La manera intencional de declarar una función *que no haga nada* pues tiene que darse con un comando *que tampoco haga nada.* Os presentamos el comando **pass.** Y eso es lo que hace, nada de nada. Mirad este código:

```
>>> def adios():  ...     pass #Aquí colocaremos código a futuro  ...  >>
> adios()  >>>
```

Como veís, al invocar la función no realiza "nada". Aquí vamos a explicar línea por línea:

- Línea 1: definimos la función con la palabra reservada **def** seguido de un espacio y el nombre que hayamos escogido para la función. Luego colocamos par de paréntesis, uno que abre y otro que cierra, allí luego colocaremos los argumentos o datos que le queremos pasar a la función. Los dos puntos ":" indican que la línea no acaba allí, que hay más en líneas siguientes y al presionar intro observaréis que el prompt pasa a ser "..." y nos invita a escribir la segunda línea.
- Línea 2: en la primera parte de este tutorial hablamos de configurar nuestro editor de textos para que al presionar la tecla TAB nos inserte espacios en vez del caracter tabulador. ***Ahora os rogamos, para cumplir con las normas de Python.org que lo configuréis exactamente a 4 espacios*** o que pulséis 4 espacios la primera vez y luego vuestro procesador indentará automáticamente cada vez que presionéis intro. Así que colocamos el comando pass (que no hace nada sino gastar 4 bytes en nuestro disco duro al estar guardado en un archivo). Luego colocamos un comentario sobre lo que pensamos hacer a futuro con dicha función.
- Línea 3: una línea en blanco para poder presionar intro y volver al prompt interactivo. *También debemos dejar una línea en blanco si estuvieramos escribiendo en un archivo, por razones de legibilidad y claridad en nuestro código.*

Creación de una función útil en la vida real.

Muy bonito estudiar la teoría pero debemos de ser pragmáticos y ponerla a trabajar en nuestras vidas. Para ello vamos a realizar una función que dado un monto nos calcule el impuesto a las ventas que debemos pagar (Impuesto al Valor Agregado, I.V.A. o IVA). En nuestro país, Venezuela, todo eso está normado por la ley y si queréis revisarlo nosotros hemos publicado una

entrada al respecto.

El cálculo es el siguiente: al monto (que llamaremos base gravada o **base**) le multiplicamos por la tasa de impuesto y la dividimos entre cien. A la fecha la tasa es de un 12% pero puede aumentar o disminuir con el paso del tiempo, pero en general dura meses y hasta años en un mismo valor. Asumiremos entonces que es 12% y pasamos a escribir la función:

```
>>> def impuesto(base):  ...     print('Impuesto a pagar:', round(base*12
/100,2))  ...   >>> impuesto(100)  Impuesto a pagar: 12  >>>
```

Rápidamente observamos, en la única linea de la función, como hemos utilizado la función **print()** , hemos separados con comas los argumentos para nos imprima un espacio en blanco entre ellos, y uno de los argumentos le hemos pasado una funcion **round()** a su vez con dos argumentos: el cálculo del impuesto en sí y el número 2 para indicarle que redondee el impuesto a dos decimales ya que así lo exige la ley (hay ciertos casos como el manejo de moneda extranjera o divisa -por ej.: €- y venta de hidrocarburos que, debidos a los altos montos, al multiplicar con solo 2 decimales trae mucha diferencia: solo en esos casos se exigen 4 decimales *¿por qué demonios explicamos esto? Más adelante veréis*).

Por supuesto, el comando pass lo eliminamos y el comentario también, pero no debemos dejarlo así, siempre debemos agregarle un texto explicativo, no solo para nosotros a futuro, que se nos pueden olvidar las cosas (¡vamos , a menos que seas Rainman!) sino para enriquecer a la comunidad de software libre, debemos compartir nuestras creaciones (y cobrar por el servicio, claro está, el que trabaja se gana su pan diario). *Si con esos argumentos no os hemos convencido aquí va un tercero que es necesario:* Python.org tiene algo llamado **docustring** que fue diseñado para que se puedan construir manuales a partir de un formato especial que nosotros agreguemos a nuestras funciones. ¿Cómo funciona esto? Solamente os decimos que debemos seguir estudiando cómo escribir funciones y en su oportunidad volveremos a tocar el tema.

Baste entonces con escribir nuestro comentario encerrado en un entrecomillado triple para cumplir con Python.org y sus normas, veamos

```
>>> def impuesto(base):  ...     ''' Cálculo del IVA a un monto dado '''
 ...     print('Impuesto a pagar:',round(base*12/100,2))  ...   >>>
```

¿A que es más lúcida nuestra función? Pero ojo, ahora vamos a llamar la función **impuesto()** tal cual, sin ningún monto y veremos un hermoso error, al traste con nuestra emoción, pero la computadora está en lo correcto, y si nosotros hemos fallado ¿qué esperamos entonces de

nuestros futuros usuarios de nuestros programas?

Pues he aquí que vamos a mejorar nuestra función, la versión 0.1, de esta manera:

```
>>> def impuesto(base=0):  ...     ''' Cálculo del IVA a un monto dado V.
 0.1 '''  ...     if base==0:  ...         print('Introduzca un monto may
or que cero.')  ...     else:  ...         print('Impuesto a pagar:', rou
nd(base*12/100,2))  >>>
```

Agregamos un valor por defecto así que si al llamar a la función no le pasamos argumento entonces toma un valor de cero y si el valor es cero imprime un mensaje invitando a hacerlo. Hemos convertido una función con argumento obligatorio en una que no lo es. Esto tiene ventajas y desventajas: si no le ponemos valor por defecto *nos obligamos nosotros mismos a ser mejores programadores.* Pero como le pusimos valor por defecto es recomendable que el programa avise que se le está pasando, así sea verdad o no, un valor cero, por lo tanto el impuesto sería cero y la función no tendría razón de ser. Si nos equivocaramos programando y en algún punto llamamos a la función sin argumento con el tiempo recibiremos una hermosa llamada telefónica de nuestros clientes diciendo algo como "metemos el monto de la base y el programa dice que introduzcamos un valor mayor que cero".

Así nuestra función sigue evolucionando: volvemos a colocarla para que sea obligatoria la base como argumento PERO dejamos el mensaje y lo mejoramos para manejar valores negativos, la versión 0.2 quedaría así:

```
>>> def impuesto(base):  ...     ''' Cálculo del IVA a un monto dado Ver.
 0.2 '''  ...     if base>>
```

Funciones con argumentos con palabras clave.

Os dijimos que el monto de la tasa de impuesto dura meses y hasta años con el mismo valor, pero eventualmente cambia. Por ello vamos a pensar en el futuro y modifiquemos otra vez, sería la versión 0.3:

```
>>> def impuesto(base, tasa=12):  ...     ''' Cálculo del IVA a un monto
dado Ver. 0.3 '''  ...     if base 0:  ...             tasa=float(n_tasa)
  ...         print('Impuesto a pagar: Bs.', round(base*tasa/100,2))  ...
  >>>
```

Aquí le damos la oportunidad al usuario de modificar la tasa si ha cambiado, de lo contrario pulsa intro y la tasa seguirá con su valor por defecto de un 12% que le colocamos al declarar la función. La función **input()** devuelve una cadena de texto: si el usuario escribió algo la longitud de la cadena es mayor a cero y la función len() nos lo dirá así, por otra parte la función **float()** convertirá la cadena de texto a un número de doble precisión, capaz de manejar decimales.

Pasaremos por alto que el usuario introduzca un valor no numérico, ya llegará el momento de mejorar nuestro código. Por ahora nos interesa seguir aprendiendo. Así como modificamos nuestra función nos permite para la tasa *desde la llamada a la función,* por ello es lo correcto mostrar dicho monto para que la función **input()** enseñe la tasa que va a aplicar, podemos modificar la línea de la siguiente manera:

```
...         n_tasa = input('Introduzca tasa ('+str(tasa)'%)')
```

Para llamar a la función escribiremos **impuesto(100 , 15)** para indicarle a la función, por lógica, que a un monto de Bs. 100 le calcule el 15% de impuesto. Pero con Python tenemos varias maneras de llamar a la función y todas producirán el mismo resultado: Bs. 15 -si simplemente presionamos intro cuando verifica la tasa-, probadlo en vuestras cónsolas por favor:

```
>>>                  impuesto(100                  ,                  15
       )               Impuesto       a       pagar:       Bs.       15
.0                  >>>         impuesto(base=100        ,         tasa=15
)                Impuesto        a        pagar:        Bs.        15.0
                     >>>           impuesto(tasa=15,           base=100
)                Impuesto        a        pagar:        Bs.        15.0
                        >>>            impuesto(tasa=15,            100
)    impuesto(tasa=15 , 100)                        ^  SyntaxError: non-
keyword arg after keyword arg
```

En la última entrada nos arroja error *porque no estamos pasando el valor con su palabra clave, Python espera que el primer argumento sea obligatorio y no importa si solo tiene 2 argumentos (con una palabra clave pues por descarte el otro argumento es el "restante"), es decir, si vamos a cambiar el orden de los argumentos siempre debemos pasarle la palabra clave para indicarle a donde va cada valor.* De otra manera, pues nos aprendemos de memoria el orden exacto de los argumentos de cada función. Si tenemos un editor que nos ayude a escribir las funciones, bien, por medio de *tooltips* nos dirán el nombre y tipo de variable que espera cada función mientras estamos escribiendo un comando al programar. **Pero en GNU/Linux se necesitan programas pequeños que hagan cosas pequeñas y las hagan bien.** Previendo la programación a futuro,

rumbo a la inteligencia artificial, Python nos ofrece algo muy interesante. Empecemos por **docustring** ¿lo recordáis?

Anotaciones en funciones.

Podemos 'preguntarle' a la función para qué sirve por medio de su **docustring** simplemente ingresando lo siguiente:

```
>>> print(impuesto.__doc__)   Cálculo del IVA a un monto dado V. 0.3
```

Ya vamos viendo la utilidad de documentar a medida que programamos; también tenemos la posibilidad de "saber" cuántos argumentos recibe la función sin haberle pasado un solo dato, lo que nos prepara para el poder usarla (si es que otra persona la escribió *o es lo que nosotros escribimos para otras personas*). Para ello debemos agregar a nuestra función la siguiente notación " **-> float:** " ya que así definimos qué tipo de dato espera entregar y recibir la función (eso disciplina nuestras tácticas de programación, pasandole a la función el tipo correcto de dato):

```
>>> def impuesto(base: float, tasa: float =12) -> float:  ...     ''' Cál
culo del IVA a un monto dado Ver. 0.3 '''  ...     if base 0:  ...
  tasa=float(n_tasa)  ...     print('Impuesto a pagar: Bs.', round(base*t
asa/100,2))  ...  >>>print(impuesto.__annotations__)  {'base': , 'return'
: , 'tasa': }
```

¿Observáis el formato como devuelve las definiciones de los argumentos? Vamos a analizarlos y para ello lo colocaremos en varias líneas de manera indentada:

```
{      'base': ,      'return': ,      'tasa':   }
```

La primera y última línea encierra la definición por medio de corchetes de apertura y cierra. La segunda línea indica que un argumento tiene como nombre clave la palabra 'base' y debe ser un número flotante. La tercera línea "return' indica lo que devuelve la función, una variable numérica flotante (la cual habremos redondeado a dos decimales, como explicamos). No especifica nombre pues ya sabemos cómo se llama la función. La cuarta línea también indica que debe recibir un argumento llamado 'tasa' y debe ser númerica de tipo flotante. Notarán que las nombran entre corchetes angulares como "class" o clase en castellano: si nosotros hiciéramos nuestras propias clases podremos referirnos a ellas en cualquier momento. Hay clases definidas de antamano en

Python: 'float' es una variables numérica que admite hasta 16 decimales (en su momento veremos los tipos de datos disponibles en Python).

Las anotaciones de funciones son totalmente opcionales y están definidas en PEP 484 y son de reciente adaptación, año 2015, aunque tienen su base en otras PEP, aparte de ser compatibles, en cierta forma.

Palabra clave "*return*".

Lo próximo que vamos a realizar es transformar completamente nuestro código a una verdadera función con el comando **return**, veamos:

```
>>> def impuesto(base: float, tasa: float =12, decimales: int =2) -> floa
t:  ...     ''' Cálculo del IVA a un monto dado Ver. 0.4'''  ...     mont
o=round(base*tasa/100, decimales)  ...     return monto  ...   >>> valor_
impuesto=impuesto(100)  >>> print(valor_impuesto)  12.0
```

Lo que hicimos fue agregarle un argumento adicional, opcional, con la palabra clave "decimales" y que por defecto es 2 (por ley de nuestra República) y la "tasa" de impuesto que por ahora es 12 pero que le podemos pasar a la función un valor diferente sea el caso. También utilizamos una variable llamada "valor_impuesto" donde guardamos el valor devuetlo para la función y a continuación lo imprimimos por pantalla; pero esta variable la podemos usar donde y como la necesitemos. Así podemos practicar de varias maneras el por palabras claves, veamos, que la práctica hace al maestro (**recordemos que estamos en modo interactivo y simplemente al llamar la función se imprime automáticamente por pantalla, pero si la guardamos en un archivo .py debemos guardar en una variable para luego imprimirla, tal como codificamos, de lo contrario no tendremos salida por pantalla**):

```
>>> impuesto(20)  2.4  >>> impuesto(100, tasa=7)  7.0  >>> impuesto(100,
tasa=15, decimales=4)  15.0  >>> impuesto(1477, tasa=17, decimales=4)  25
1.09  >>> impuesto(1970, tasa=17.89, decimales=4)  352.433  >>> impuesto(
1341, decimales=4, tasa=17.89)  239.9049
```

Lo que siempre debemos hacer es "pasarle" la base, y luego por medio de palabras claves, sin importar el orden, los otros dos argumentos, de lo contrario arroja error (o excepción que es el nombre correcto). Si queremos pasarle argumentos, sin importar el orden, debemos colocarle palabras claves a todos los argumentos, de la siguiente manera:

```
>>> impuesto(decimales=2, tasa=10, base = 1500)  150.0  >>> impuesto(deci
males=2, tasa=10)  Traceback (most recent call last):   File "/usr/lib/x8
6_64-linux-gnu/gedit/plugins/pythonconsole/console.py", line 378, in __ru
n   r = eval(command, self.namespace, self.namespace)   File "", line 1,
in   TypeError: impuesto() missing 1 required positional argument: 'base'
```

Sin embargo, si le pasamos solamente los decimales y la tasa *y no le pasamos la base, arroja error,* porque ese debe ser un argumento obligatorio , ya que tasa y decimales le colocamos valores por defecto y así los convertimos en argumentos opcionales.

Uso de variables para pasarlas a las funciones.

Veremos algo que es facilmente deducible, pero lo complicaremos un poco con los valores por defecto en los argumentos opcionales de una función. Nos explicamos: podemos guardar en variables los valores y luego pasarlos a la función:

```
>>> miMonto=450  >>> miTasa=10  >>> misDecimales=3  >>> miImpuesto=impues
to(miMonto, miTasa, misDecimales)  >>> print(miImpuesto)  45.0  >>> miImp
uesto=impuesto(tasa=miTasa, base=miMonto, decimales=misDecimales)  >>> pr
int(miImpuesto)  45.0
```

Esto no necesita mayor explicación: es simplemente pasarle los valores por medio de variables. Lo que es más avanzado es pasarle los valores por defecto a la función por medio de variables *declaradas antes de definir la función* y dichas variables luego pueden cambiar su valor *pero la función quedará definida con los valores dados exactamente antes de haber definido la función.* Podemos así redefenir la función y luego explicaremos su utilidad:

```
valor_tasa=15  def impuesto(base: float, tasa: float =valor_tasa, decimal
es: int =2) -> float:      ''' Cálculo del IVA a un monto dado Ver. 0.4''
'      monto=round(base*tasa/100,decimales)      return monto    valor_ta
sa=16
```

Aunque luego cambiemos la variable **valor_tasa** a 16, la función **impuesto()** quedará en un valor de por defecto de 15 en la tasa a lo largo de todo el programa.

¿Recordáis que os dijimos que la tasa de impuesto puede variar, según las necesidades fiscales del país? Pues bueno, nuestra función se puede preparar a futuro (por ejemplo, realizamos una sencilla aplicación de facturación y manejo de inventario) *si escribimos nuestras funciones de esta manera, las podemos a volver a recompilar rápidamente si al módulo principal le cambiamos el valor.*

Funciones con argumentos arbitrarios.

Podemos definir una función que acepte cualquier argumento, habiendo ya estudiado anteriormente la flexibilidad que denota Python. Son pocos caracteres que la componen, pero no se dejen engañar: es una función compleja y difícil de asimilar, muy genérica, y es difícil darle un uso práctico:

```
>>> def mi_funcion(*argumentos):  ...     for mi_arg in argumentos:  ...
        print(mi_arg)  ...   >>> mi_funcion(20)  20  >>> mi_funcion()  >>
> mi_funcion(20,"prueba")  20  prueba  >>> mi_funcion(20, "prueba", 17, 2
0, "otra prueba")  20  prueba  17  20  otra prueba  >>>
```

Funciones con argumentos arbitrarios pero con palabras claves.

Para "declarar" argumentos arbitrarios en una función solamente tenemos que precederla con un asterisco y así Python "sabrá" agruparlas en un "diccionario" lo cual nos permite listarlas (y trabajar) dichos argumentos (que serán una simple variable más dentro de la función). Pero como mejor es explícito que implícito, Python nos permite pasar argumentos con palabras claves (y todas deben llevar su palabra clave o producirá una "errror" o excepción). *Tomemos de nuevo nuestra función de cálculo de IVA con una base dada*

```
>>> def impuesto(**argumentos) -> float:  ...     ''' Funciones con argum
entos y palabras claves '''  ...     monto=0  ...     base = 0  ...     t
asa = 12  ...     decimales = 2  ...     for arg_clav in argumentos.keys(
):  ...         if arg_clav=='base':  ...             base=float(argument
os[arg_clav])  ...         if arg_clav=='tasa':  ...             tasa=flo
at(argumentos[arg_clav])  ...         if arg_clav=='decimales':  ...
        decimales=int(argumentos[arg_clav])  ...     monto=round(base*tas
a/100,decimales)  ...     return monto  ...  >>> monto_iva=impuesto(decim
ales=2, base=500, tasa=7)  >>> print(monto_iva)  35.0  >>> print(impuesto
.__annotations__)  {'return': }  >>> print(impuesto.__doc__)   Funciones
con argumentos y palabras claves   >>>
```

Ya nuestras funciones van ganando complejidad y cada vez es más difícil escribirlas en modo interactivo, creemos que ya vale la pena escribirlas en un archivo por medio de nuestro editor de texto favorito, pronto daremos unas recomendaciones al respecto y unas normativas de Python. Mientras, veámos que sucede en este último ejemplo:

- Línea 1: declaramos la función en sí con metadatos incluídos.
- Línea 2: le establecemos su **docustring**.
- Líneas 3 a 6: incializamos las variables con su valor pro defecto, si lo tuviera.
- Línea 7: llamamos a un ciclo for para enumerar todos los argumentos con palabra claves "pasadas" a la función.
- Líneas 8 a 13: por medio de sentencias condicionales verificamos si todos los elementos de la fórmula del cálculo del impuesto están presentes, si faltare alguno pues tomaría su valor por defecto.
- Línea 14: realiza el cálculo en sí.
- Línea 15: devuelve el resultado hacia fuera de la función.
- Demás líneas: anotaciones de la función y su **docustring**. Obsérvese que en las anotaciones solo nos indica que es una función que solo devuelve un valor numérico flotante, sin ninguna información de los argumentos necesarios para que "haga su trabajo"

¿Qué utilidad tiene esto, declarar una función tan extraña que acepte cualquier cantidad de argumentos con palabras clave? Cuando veamos los *diccionarios* en mayor detalle se nos ocurrirán unos cuantos usos útiles, en este punto vamos a probar algo muy sencillo:

```
>>> diccionario = { "base" : 277.25 , "tasa" : 19 , "decimales": 3 }  >>>
 impuesto(**diccionario)  52.678  >>>
```

Fíjense que las palabras claves deben estar encerradas entre comillas, y que debemos colocarle dos asteriscos al llamar a la función, ***así estemos declarando la función con **argumentos, esto es lo más curioso del asunto y nos devuelve al principio:*** cualquier función acepta los argumentos con palabras claves si le colocamos ** al llamar la función con un diccionario *¿complejo, cierto? Y apenas estamos comenzando ;-) .*

Funciones "clonadas".

Python nos permite tomar las funciones que importemos y "cambiarles" el nombre. Así nuestra función impuesto() que escribimos, tal vez sea para nosotros memorizarla de otra manera, por ejemplo impt(), pues solo debemos escribir impt = impuesto y pasarle los valores con el diccionario que hicimos anteriormente, he aquí:

```
>>> diccionario = { "base" : 277.25 , "tasa" : 17 , "decimales": 4 }  >>>
```

```
impt = impuesto  >>> impt(**diccionario)  47.1325  >>>
```

Función abreviada lambda.

A veces necesitamos usar una función una sola vez (cosa rara) y codificarlo en 3 líneas como mínimo (aparte de ir en una sección aparte de nuestro proyecto dedicada a las funciones) pues que nos lleva tiempo y espacio. Para ello echamos mano de la ¿función de funciones? el comando **lambda**. Este comando nos permite aplicar, por ejemplo, un cálculo como el nuestro, que es sumamente sencillo, practiquemos:

```
>>> (lambda base : round(base*12/100,2))(100)  12.0
```

Como vemos no le colocamos nombre a la función, solo le indicamos que devuelva la misma base con el cálculo del impuesto (es todo lo que encierra el primer par de paréntesis de izquierda a derecha) y en el segundo par de paréntesis le pasamos el argumento con la base a calcular. Tal vez sea complicado de entender así que pasamos a definir la función y "clonarla" con un nombre:

```
>>> impuesto=lambda base : round(base*12/100,2)  >>> impuesto(1000)  120.
0  >>> impuesto(7)  0.84  >>> impuesto(70)  8.4  >>> impuesto(100)  12.0
 >>>
```

Acá vemos que es una manera de definir rápidamente una función. Si queremos colocarle los argumentos completos la podemos definir de la siguiente manera:

```
>>> impuesto=lambda base, tasa, decimales: round(base*tasa/100,decimales)
  >>> impuesto(100,12,2)  12.0  >>> impuesto(543,17,4)  92.31  >>>
```

Evidentemente que los argumentos son posicionales, si queremos agregarles palabras claves... *pues para eso definimos la función completa como ya lo hemos hecho.* Consideramos útil a este comando/función dentro de otra función declarada para que se circunscriba dentro de esa función solamente, pero vosotros juzgad y dadle algún otro uso novedoso, ¡a practicar!

Estilo al codificar.

En PEP 8 se especifica cómo codificar nuestras funciones y para que queden correctas deben cumplir con las suguientes especificaciones:

- Se deben indentar con 4 espacios y sin tabuladores (ver al final nuestras recomendaciones).
- Las líneas no deben superar los 79 caracteres.
- Se deben usar líneas en blanco entre funciones y clases.
- Cuando sea posible, coloque los comentarios en la misma línea que describe.
- Se debe usar **docustring** (¿deberíamos traducirlo como '**docutexto**'?).
- Se deben usar espacios entre los operadores y después de las comas *pero no dentro de los paréntesis,* ejemplo: **a = f(1,2) + g(3,4)** .
- En el caso de las funciones y métodos se deben separar las palabras con guión bajo, por ejemplo **impuesto_iva()** , **impuesto_sobre_renta()** ; las clases se deben nombrar en estilo jorobas de camello: **TasaDeImpuesto**.
- En el caso de que nuestro código sea para uso internacional, deberemos usar solamente caracteres básicos. Evidentemente que nuestro idioma utiliza muchos caracteres extendidos en UTF-8, así que todo dependen hacia adonde va dirigido.

Uso de gedit con python.

Por mucho nuestro editor de texto favorito es **gedit** debido a que tiene ciertas características deseables para programar pequeños proyectos de código.

Las capturas de pantalla aquí mostradas corresponde a gedit 3.10.4:

Al ejecutarlo lo primero que debemos activar es la barra de herramientas y la barra de estado en el menú desplegable "Ver":

Al marcar estas dos opciones podremos ver en la aprte superios los comandos más usados como: documento nuevo, abrir documento, guardar, imprimir, etcétera. A pesar que todos estos comandos tiene sus atajos de teclados, los íconos de los botones tienen un efecto muy intuitivo. La parte más importante es la parte inferior la cual permite escoger qué lenguaje estamos programando y justo al lado la cantidad de espacios a insertar en el indentado automático y al presionar la tecla tabulador inserte espacios en vez de la caracter tabulador en sí, veamos:

También podemos guardar nuestras perferencias para la próxima vez que ejecutemos gedit, para hacerlo vamos al menu desplegable "Editar" -> "Preferencias" y marcamos las siguientes

opciones, todo según las normas de estilo de Python:

- Mostrar los números de línea nos sirve para hallar rápidamente dónde debemos corregir cuando el depurador Python nos indica algún fallo de sintaxis.
- Al mostrar el margen derecho con una pequeña marca de agua nos ayuda a no sobrepasar los 79 caracteres.
- Si activamos el ajuste de texto, así no sobrepasamos los 79 caracteres por línea *pero cambiamos el tamaño de nuestra ventana,* gedit nos colocará las líneas que no quepan en la ventan en múltiples líneas *pero respetando la numeración para indicarnos que es una sola línea sobre la que estamos trabajando.* La opción de no dividir palabras nos permite los comandos completos, sin divisiones.
- La opción de resaltar la línea actual nos permite enfocarnos en la linea sobre la cual estamos editando y si es multilínea (ve punto anterior) nos ofrece un mejor panorama.
- Por último resaltar parejas de corchetes nos permite anidar funciones y sus argumentos son más fáciles de visualizar.

Una vez hayamos fijado nuestras preferencias (la cuales se aplican en tiempo real, así que muestran como se ve nuestro archivo de una vez) también podemos activar los complementos que nos serán muy útiles:

- El completado de palabras "memoriza" los comandos que hayamos escrito y al nosotros volver a escribirlos el complemento lo sugiere por medio de un pequeño menú emergente. Recomendamos hacer click en "preferences" y marcar interactivo y un mínimo de 3 letras por palabra.
- Completar paréntesis, corchetes rectos y llaves evita el error común de no hacerles el cierre. No funciona con los corchetes angulares "".
- Por último la cónsola python: al activarla podemos pulsar CTRL+F9 y una subventana en la parte inferior nos abre una cónsolar interactiva con práticamente las mismas características de la cónsola python. Podemos volver a la edición del documento con la tecla ESC y volver a abrir la cónsola python con CONTROL+F9.

Fuentes consultadas:

En idioma castellano:

- "Programa como un Pythonista: Python Idiomático" por David Goodger.

Enlaces en idioma inglés:

- "Clearing the Console Screen" by Kent Johnson.
- «Editing PYTHONPATH (or "Where's my module?!")» by Joe Filippazzo.
- "Using lambda functions" at DiveIntPython.
- "Importing Python Modules" by Fredrik Lundh.