Funciones en Python

Apuntes versión 1.0

Víctor Araya Sánchez

2017

Objetivos de este apunte

Al terminar de leer, hacer los ejercicios propuestos de este apunte y trabajar en clases serás capaz de:

- Utilizar funciones nativas e importadas del lenguaje de programación Python con un objetivo concreto y acotado.
- Crear funciones en el lenguaje de programación Python para resolver distintas situaciones acotadas.

1. Introducción

El presente apunte tiene como fin que aprendas a utilizar las funciones de Python como herramientas prácticas para la resolución de distintos tipos de problemas.

Para lograr esto partiremos por lo básico... ¿Qué es una función?

función

Del lat. *functio, -ōnis.*

1. f. Capacidad de actuar propia de los seres vivos y de sus órganos, y de las máquinas o instrumentos.

2. f. Tarea que corresponde realizar a una institución o entidad, o a sus órganos o personas.

3. f. Acto solemne, especialmente el religioso.

4. f. Representación de un espectáculo, especialmente teatral, o proyección de una película. U. t. en sent. fig.

5. f. Obra teatral representada o película proyectada.

6. f. Fiesta mayor de un pueblo o festejo particular de ella.

7. f. Convite obligado de los mozos.

8. f. Escándalo o alboroto que se produce en una reunión.

9. f. *Ling.* Papel relacional que, en la estructura gramatical de la oración, desempeña un elemento fónico, morfológico, léxico o sintagmático.

10. f. *Ling.* Finalidad de los mensajes verbales. *La función expresiva del lenguaje.*

11. f. *Mat.* Relación entre dos conjuntos que asigna a cada elemento del primero un elemento del segundo o ninguno.

12. f. *Mil.* Acción de guerra.

Haremos énfasis en las dos primeras acepciones que aparecen en el cuadro anterior: Las funciones en programación corresponden a pequeños[1] tramos de código con una tarea específica.

Particularmente en nuestro caso consideraremos además que una función siempre deberá entregar alguna información luego de ejecutarse.

En Python existen una gran cantidad de funciones predefinidas y listas para utilizarse. A este grupo de funciones les llamaremos **Funciones Nativas**, dado que son propias del lenguaje de programación.

Existirán muchos casos en los que las funciones nativas no basten por si solas para solucionar un problema, favorablemente para esos casos Python nos permite definir nuestras propias funciones.

Adicionalmente, tanto desarrolladores como usuarios de Python han desarrollado un sinnúmero de funciones, las que se agrupan en paquetes específicos o **Módulos** y que pueden llamarse o importarse a nuestro código para apoyar el proceso de programar una solución. A estas últimas las llamaremos **Funciones Importadas**.

2. Funciones Nativas

Para conocer todas las funciones nativas que tiene Python, y saber qué hace cada una de ellas, debes revisar el siguiente enlace: Link.

Actividad

¿Para qué sirven las funciones `abs()`, `round()` y `pow()` ?

Como habrán notado al revisar el enlace, la mayoría (o casi todas) de las funciones expuestas requieren de alguna *entrada* con la que trabajar.

Ejemplo

```
>>> len("murcielago")
10
```

Una analogía comúnmente utilizada para explicar una función en programación es pensar en las funciones matemáticas, las cuales consideran:

[1] La extensión de una función puede variar dependiendo de lo compleja que sea la tarea específica que cumple.

- Un conjunto de origen – Entrada
- Una relación – Procesamiento
- Un conjunto de llegada – Salida

Para el ejemplo el conjunto de origen corresponde al conjunto de los strings, la relación len se preocupa de contar la cantidad de caracteres que hay en la entrada y el conjunto de salida corresponde los enteros, dado que entrega un valor equivalente a la cantidad de caracteres que tiene el string murcielago.

En el ejemplo, la función `len()` requiere de una *entrada* iterable[2], en este caso un string que corresponde a la palabra *murcielago*. A esta *entrada* le llamaremos **entrada**, **argumento** o **parámetro actual**.

Al proceso de usar una función, sea nativa, importada o creada, le llamaremos **invocación**.

Algunas funciones requieren de argumentos obligatorios, escritos en la documentación entre paréntesis redondos `()` y argumentos opcionales, escritos entre corchetes `[]` . Un ejemplo de estos es la función `range(start, stop[, step])` . Esta referencia es exclusiva para la documentación, dado que toda función utilizada recibe sus argumentos a través de paréntesis `()` y en caso de no requerirlos estos se colocan vacíos para que el intérprete la reconozca como tal.

Ejemplo

```
>>> range(1, 6)
[1, 2, 3, 4, 5]
>>> range(0, 20, 5)
[0, 5, 10, 15]
```

`range()` genera una lista de números enteros que van desde `start` a `stop`, sin considerarlo. `step` es un parámetro actual opcional, y determina la diferencia entre dos elementos consecutivos de la lista que deseas generar, es decir, el paso. Omitir `step` equivale a darle valor 1.

Dentro de estas funciones nativas existen algunas especializadas en cambiar los tipos de datos con los que trabajamos, ya sean `(int)` , enteros largos `(long)` , números de punto flotante `(float)` , booleanos `(bool)` , listas `(list)` y strings `(str)` :

[2]Se entiende por iterable todo tipo de datos que se pueda recorrer como listas y strings. No obstante, Python cuenta con otros objetos iterables como las tuplas, conjuntos y diccionarios.

Nombre	Descripción
`int(x)`	Recibe un número o un `string` de caracteres numéricos y lo convierte a un entero, si el número no puede ser contenido en 4 bytes, se guarda como `long` , en caso de recibir un `float` , el resultado se trunca, no se aproxima.
`long(x)`	Recibe un número o un `string` de caracteres numéricos y lo convierte a un entero largo, (sin importar que pueda ser representado como `int`), en caso de recibir un `float` , el resultado se trunca, no se aproxima.
`float(x)`	Recibe un número o un `string` de caracteres numéricos y lo convierte a un flotante.
`bool(x)`	Recibe un dato e intenta su conversión a booleano, si el dato está vacío entrega `False` , en caso contrario retorna `True` .
`list(x)`	Recibe un tipo de dato iterable y lo convierte a una lista, en caso de no recibir argumentos, entrega una lista vacía.
`str(x)`	Recibe un dato y lo convierte a `string` .

Actividad

Prueba las funciones de cambio de tipo en el intérprete de Python y responde las siguientes preguntas:

- ¿Qué ocurre cuando se utiliza `list()` con un `int` , `long` , `float` o `bool` ?
- Si x es un tipo de dato `int` ¿para qué valores de x la función `bool(x)` retorna `False` ? Repita la pregunta para x siendo un tipo de dato `long` , `float` , `list` y `string` .
- ¿Qué ocurre con la función `str()` si recibe como entrada una expresión matemática?
- ¿Qué ocurre con la función `int()` si recibe una lista o un string con caracteres distintos a dígitos?
- ¿Qué ocurre con la función `float()` si la entrada es `2+5**2` ?

Otras funciones nativas que hemos utilizado durante el semestre han sido las funciones:

- `input()` : Muestra el mensaje de entrada en el intérprete de Python, y queda a la espera de una expresión que ingrese el usuario vía entrada estándar (teclado), evalúa la expresión y retorna el resultado.
  - Entrada: Mensaje (`string`) que se mostrará al usuario en el intérprete de Python.
  - Salida: Resultado de la expresión evaluada / Error si la expresión no es válida.
- `raw_input()` : Muestra el mensaje de entrada en el intérprete de Python, y queda a la espera de un `string` que ingrese el usuario vía entrada estándar (teclado), una vez recibido el `string` , lo retorna.

- Entrada: Mensaje (`string`) que se mostrará al usuario en el intérprete de Python.
- Salida: `string` ingresado por el usuario.

Ejemplo

Supongamos que necesitamos un programa que debe mostrar la cantidad de cifras de un número que ingrese el usuario.

Lo primero que debemos hacer, antes de comenzar a escribir código, es pensar en lo que debemos lograr y analizar que herramientas disponibles tenemos:

- Necesitamos que el usuario ingrese un número, y para ello tenemos disponible la función `input()`.
- También necesitamos contar la cantidad de cifras de un número. Esta tarea no es directa con ninguna de las funciones que tenemos disponibles, dado que `len()` cuenta la cantidad de elementos de un objeto iterable y los datos de tipo numérico (`int`, `long`, `float`) no lo son. No obstante, podemos transformar dichos números en datos de tipo `string`, los que sí son iterables.

Con lo anterior en mente podemos resolver el problema como sigue:

```
# -*- coding: cp1252 -*-
# Permite el uso de tildes y ñ

# Solicitamos el número al usuario:
numero = input("Ingrese un número: ")

# Transformamos el número a un str:
numeroCaracteres = str(numero)

# Calculamos el largo del str, que equivale al número de cifras:
largoNumero = len(numeroCaracteres)

# Mostramos por pantalla el número de cifras
print largoNumero
```

Necesitamos hacer una observación del código anterior. La primera línea del programa contiene el texto `#-*- coding: cp1252 -*-`, que no es un comentario para aumentar la legibilidad del programa, sino que una indicación al intérprete de Python de que el programa usa un juego de caracteres con tildes. Esto es necesario para que nuestros mensajes se desplieguen correctamente en pantalla y funciona en el sistema operativo Windows™. En otros sistemas operativos y en algunas versiones de Windows esto generalmente se consigue escribiendo el texto `# -*- coding: utf-8 -*-` en la primera línea del programa e indicando al editor que se esté usando que guarde los archivos ".py" con este juego de caracteres.

Actividad

Parte 1
La solución al problema anterior funciona bien en el caso de números de tipo int, pero ¿qué pasa con números tipo long y float? ¿y con números negativos?

Implementa cambios en el código que solucionen los posibles problemas con dichos tipos de datos.

Parte 2
Implemente mejoras a la solución que verifiquen que lo que el usuario ingresó fue un número escrito en cifras y no palabras.

3. Funciones Importadas

Las funciones nativas corresponden a herramientas básicas de gran utilidad, sin embargo, es muy probable que no baste con ellas para resolver problemas de gran envergadura. Por lo anterior también es importante aprender a trabajar con **Módulos** externos.

Supón que necesitas trabajar con funciones trigonométricas como seno y coseno, las cuales no vienen definidas nativamente en Pyhton. Una alternativa sería definir personalmente dichas funciones, lo que podría tomar mucho tiempo. Una segunda alternativa es utilizar las que otras personas ya han definido en el módulo **math**.

A pesar que dichas funciones ya están definidas en **math**, no están disponibles directamente en Python.

Ejemplo

```
>>> sin(0)

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    sin(0)
NameError: name 'sin' is not defined
>>>
```

Para que podamos utilizar las funciones que requieres debes explicitar a Python desde dónde quieres sacarlas; en este caso, del módulo **math**. A continuación, se muestran distintas formas de dar la referencia de origen de dichas funciones a Python:

- Forma 1: `import módulo`
  - De esta forma se puede utilizar todas las funciones del módulo importado siempre que se haga referencia al módulo de origen de la función utilizando la sintaxis:
    `modulo.función(argumento)`

Ejemplo

```
>>> import math
>>> math.sin(0)
0.0
>>>
>>> math.cos(0)
1.0
>>>
```

- Forma 2: `import módulo as alias`
  - Aquí se utiliza un alias o nombre-corto para el módulo importado. De esta forma también se puede utilizar todas las funciones del módulo importado siempre que se haga referencia al alias del módulo de origen de la función utilizando la sintaxis:
    `alias.función(argumento)`

Ejemplo

```
>>> import math as m
>>> m.sin(0)
0.0
>>> m.cos(0)
1.0
>>>
```

- Forma 3: `from módulo import función`
  - Esta forma de importar no requiere de hacer referencia al módulo de origen, pero pueden utilizarse directamente solo las funciones que fueron listadas por coma al momento de hacer la importación.

Ejemplo

```
>>> from math import sin, cos
>>> sin(0)
0.0
>>> cos(0)
1.0
>>>
```

- Forma 4: `from módulo import *`
  - Esta forma de importar no requiere de hacer referencia al módulo de origen y se pueden utilizar directamente todas las funciones del módulo importado.

Ejemplo

```
>>> from math import *
>>> sin(0)
0.0
>>> cos(0)
1.0
>>>
```

A efectos de este curso, y por buenas prácticas de programación, solo se considerarán válidas las opciones de importación 1 y 2, dado que la 3 y 4 se prestan para confusiones complejas al no referenciar al módulo de origen.

En general se debe utilizar la forma 1, salvo que el nombre del módulo sea muy extenso o complejo, en cuyo caso puede utilizarse sin problemas la forma 2[3].

[3] Lo veremos en práctica más adelante con el uso del módulo `matplotlib.pyplot`.

De forma análoga es posible importar no solo funciones, sino también variables como pi:

Ejemplo

```
>>> import math
>>> math.pi
3.141592653589793
>>>
```

El módulo math, es uno de los módulos que vienen preinstalados junto con Python y componen la librería estándar de Python, pues corresponden a módulos que ayudan a resolver problemas comunes de programación. Sin embargo, como veremos más adelante en el curso, es posible instalar módulos adicionales para aumentar aún más las capacidades de Python.

Actividad

Otro módulo interesante de la librería estándar de Python es el módulo random, que permite la generación de números pseudoaleatorios[4]. Utilicemos Python para probar lo que hacen las siguientes funciones del módulo random:

- `randint(x, y)`, con `x` e `y` enteros
- `choice(x)`, con `x` siendo un elemento iterable
- `randrange(x)`, con `x` siendo un entero
- `randrange(x,y,z)`, con `x`, `y` y `z` enteros
- `shuffle(x)`, con `x` siendo una lista
- `random()`

Si tienes problemas entendiendo el funcionamiento de las funciones, recuerda que puedes consultar la documentación del módulo random de Python: Link

[4]Como el computador es una máquina sin las capacidades cognitivas de una persona y sólo es capaz de responder a órdenes, es imposible para él generar números aleatorios. Lo que en la práctica se hace es utilizar un conjunto de cálculos intrincados, a menudo usando valores de entrada ocultos como la fecha y la hora, tan difíciles de seguir que finalmente derivan en una salida muy difícil de predecir a priori.

Ejercicio Resuelto

Supongamos ahora que nuestro programa anterior requiere que en caso que el usuario ingrese una "h" en vez de un número se redirija al usuario a la página web con el manual del programa. Link: https://goo.gl/DwYwt5.

Nuevamente, antes de proceder a escribir el código de golpe nos detendremos a pensar en el problema que se presenta:

- Necesitamos abrir una página web con el link del manual del programa y no existe ninguna función nativa que haga esa tarea. Buscamos dentro de la documentación de las librerías de Python y encontramos un módulo que puede ser de utilidad: **`webbrowser`** .
- Leemos la documentación del módulo y detectamos una función que abre el navegador con una dirección específica: **`open(dirección)`** .

Con lo anterior en mente podemos resolver el problema como sigue:

```
# -*- coding: cp1252 -*-
# Permite el uso de tildes y ñ

# Importamos el módulo que permite manejar el navegador web:
import webbrowser

# Solicitamos el número al usuario directamente como str:
numero = raw_input("Ingrese un número: ")

# Ahora decidimos que hacer según lo que el usuario ingresara:
# Si el usuario ingresó una "h" debemos dirigirlo a la página del
    manual:
if numero == "h":
    webbrowser.open("https://goo.gl/DwYwt5")
# Si no, posiblemente haya ingresado un número, por lo que
   realizamos el mismo cálculo anterior:
else:
    # Calculamos el largo del str, que equivale al número de
       cifras:
    largoNumero = len(numero)

    # Mostramos por pantalla el número de cifras
    print largoNumero
```

Actividad

Parte 1
¿Qué pasaría si en el código anterior se mantuviera la entrada de datos a través de input en vez de usar `raw_input` ?

Parte 2
Implemente mejoras a la solución anterior que permita al usuario ingresando una "a" obtener resultados aleatorios.

4. Creación de funciones

Cuando necesitamos tareas muy particulares es posible que no tengamos alternativas dentro de las funciones nativas ni las importadas. Para estos casos es que Python nos permite definir nuestras propias funciones.

Como habrás notado las funciones son útiles para tareas repetitivas o para simplificar las instrucciones dentro de un código más extenso. Es posible que en paralelo a este curso estés realizando el curso de Cálculo II, en donde es común estimar el valor de una integral a través de la integral de Riemann, por lo que sería de utilidad tener una función que pudiera hacer una estimación del valor de dicha integral.

Tomaremos como ejemplo esa necesidad para desarrollar nuestra primera función. Para crear una función personalizada se debe comenzar con la palabra reservada **`def`** , continuando como sigue:

```
def nombreDeLaFuncion(parámetros, formales, separados, por, coma):
```

Con esa línea se define el **encabezado** de la función. El nombre escogido debe seguir la convención entre programadores: el nombre debe ser indicativo de lo que hace la función y debe partir con un **verbo**; si el nombre contiene más de una palabra, estas deben escribirse con la **primera letra en mayúsculas**, similar a la forma de nombrar variables.

Siguiendo al nombre, y sin espacios en medio, van los paréntesis en donde se agregan todos los argumentos que necesitará la función para trabajar separados por comas (no es problema aquí usar espacio después de cada coma), a estos argumentos les llamaremos **parámetros formales**. Estos son los nombres que se utilizan al **interior** de la función para referirse a los argumentos que se le entregan al **invocarla** o ejecutarla. El encabezado termina con un signo **dos puntos**, indicando que ahora se comienza el cuerpo de la función.

Bajo el enunciado se define el cuerpo de la función, que corresponde al bloque de sentencias que deben ejecutarse al evaluar la función:

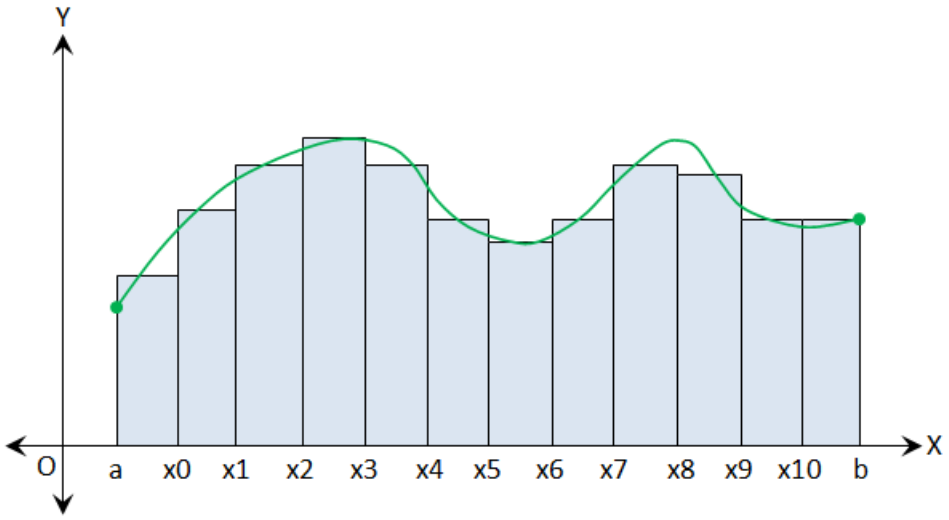
```
def nombreDeLaFuncion(parámetros,formales,separados,por,coma):
        instrucciones de la función...
        pueden ser varias líneas...
        pero siempre identadas para explicitar que son parte de la función
```

Notemos que el cuerpo de la función se escribe indentado[5], al igual que las sentencias condicionadas o dentro de bloques de iteración, respecto al encabezado de la función, es decir, tienen una sangría mayor desde el margen izquierdo. Es importante que recordemos que todas las sentencias de un mismo bloque tienen la misma indentación.

Para finalizar y marcar sintácticamente el fin de la declaración de una función, se deja una **línea en blanco sin indentar**.

Teniendo estos lineamientos en mente, crearemos la función que calcula la integral de Riemann para cualquier función matemática dada.

Tal como hemos hecho hasta ahora, antes de escribir el código buscaremos pensar y entender lo que queremos hacer. En términos prácticos, la integral de Riemann estima el área bajo la curva a través del uso de delgados rectángulos de ancho **dx** y altura equivalente al **valor de la función matemática evaluada** a lo largo del intervalo de integración.



Con esto claro, procederemos a implementar la función:

```
# Creamos la función "calcularIntegralRiemann"
# Función que calcula la Integral de Riemann de una función matemática
# cualquiera dada para un intervalo definido.
# Entrada: La función requiere de entrada:
#   - el nombre de la función matemática que se integrará - "funcion"
#   - el ancho de los delgados rectángulos - "dx"
#   - el valore del extremo izquierdo del intervalor de integración - "a"
```

[5] Anglicismo de uso común por gente que programa, ya hemos visto que la indentación se utiliza también para indicar las sentencias que quedan condicionadas a la ocurrencia de una condición if, o de un ciclo while.

```
#   - el valore del extremo derecho del intervalor de integración - "b"
# Salida: La función devuelve un float con el valor del área bajo
# la curva de la función "función" en el intervalor "(a,b)"
def calcularIntegralRiemann(funcion,dx,a,b ):
    # Se define una variable local que almacenará en lo sucesivo el área
    # que se vaya acumulando en cada paso.
    integral = 0

    # Se define una variable local que será en la cual se evalúe el
    # valor de la función matemática en cada paso.
    x = a

    # Con un ciclo se recorrerá el intervalo (a,b).
    while x < b:

        # Aquí se acumula el valor de la integral paso a paso.
        # La integra de Riemann puede calcularse utilizando para
        # un mismo intervalo el valor máximo de funcion en el
        # intervalor (x,x+dx)(Integral Superior) o el mínimo
        # (Integral Inferior).
        # Como es una estimación se optó por el punto medio del
        # ancho dx del rectángulo
        integral += eval(funcion+"("+str(x+dx/2)+")")*float(dx)

        # Se aumenta el contador para avanzar a lo largo del
        # intervalo.
        x += dx

    # Al final se agrega la instrucción "return" que permite devolver
    # el valor total calculado.
    return integral
```

Se utiliza la sentencia **`return`** para que la función devuelva el valor de la variable **`integral`** como resultado de la evaluación de la función **`calcularIntegralRiemann`** . Usar esta instrucción permite, posteriormente, realizar nuevos cálculos con el valor almacenado en dicha variable o imprimirla a pantalla según se necesite.

Es de gran importancia que notes que las variables creadas dentro de la función (integral, x) y los parámetros formales de la función (`funcion , dx , a , b`) solo existen mientras la función se ejecuta y no pueden utilizarse directamente ni después ni antes de invocar a la función.

Como ya hemos visto, por las propiedades de anidación, es posible construir funciones que internamente utilicen sentencias condicionales (`if-elif-else`) y de iteración (`while` y `for-in`). Por lo tanto, las funciones son útiles para encapsular procesos en entornos difíciles de seguir con trazas, por ejemplo, al utilizar ciclos anidados.

Una última observación está relacionada a los comentarios utilizados en la función definida, podemos ver que sobre el encabezado de la función se indica:

- Qué es lo que hace la función: en este caso, calcular la hipotenusa de un triángulo rectángulo, a partir de dos catetos.
- Qué es lo que recibe como entrada la función: indicando qué valores y a qué tipo de dato hacen referencia.

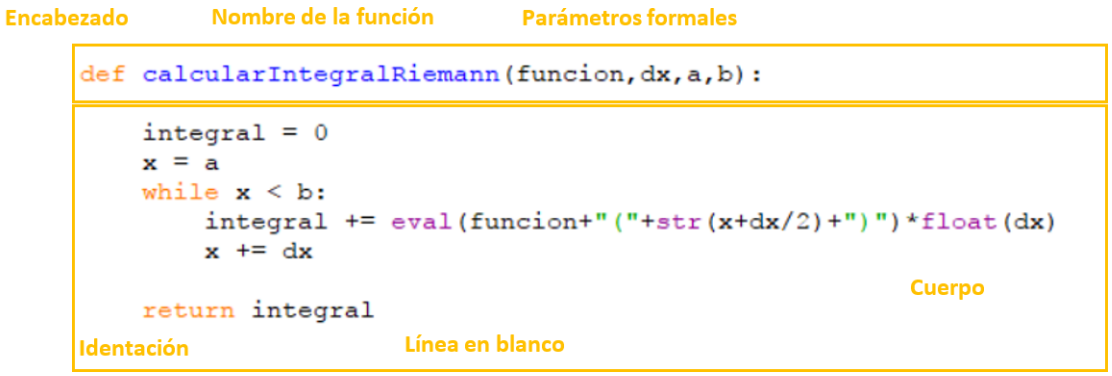
- Qué es lo que entrega como salida la función: indicando que representa la salida de la función y el tipo de dato al que pertenece.

Estos comentarios los consideraremos obligatorios a efectos de buenas prácticas, para todas las funciones, pues permiten que los programadores entiendan cuál es el proceso que la función encapsula sin tener que consultar el detalle del código.

Actividad

En la definición de la función anterior ¿qué hace la función nativa **`eval`** ?

Esquema de función:



Ejemplo

Ahora, probemos la función que acabamos de crear, en un programa que calculará la integral de Riemann de coseno entre 0 y pi/2 y su integral analítica.

```
# -*- coding: cp1252 -*-

# Importamos el módulo matemático
import math

# Creamos la función "calcularIntegralRiemann"
# Función que calcula la Integral de Riemann de una función matem
    ática
# cualquiera dada para un intervalo definido.
# Entrada: La función requiere de entrada:
#    - el nombre de la función matemática que se integrará - "
    funcion"
#    - el ancho de los delgados rectángulos - "dx"
#    - el valore del extremo izquierdo del intervalor de integraci
    ón - "a"
```

```
#   - el valore del extremo derecho del intervalor de integración
    - "b"
# Salida: La función devuelve un float con el valor del área bajo
# la curva de la función "función" en el intervalor "(a,b)"
def calcularIntegralRiemann(funcion,dx,a,b):
    # Se define una variable local que almacenará en lo sucesivo
       el área
    # que se vaya acumulando en cada paso.
    integral = 0

    # Se define una variable local que será en la cual se evalúe
       el
    # valor de la función matemática en cada paso.
    x = a

    # Con un ciclo se recorrerá el intervalo (a,b).
    while x < b:

        # Aquí se acumula el valor de la integral paso a paso.
        # La integra de Riemann puede calcularse utilizando para
        # un mismo intervalo el valor máximo de funcion en el
        # intervalor (x,x+dx)(Integral Superior) o el mínimo
        # (Integral Inferior).
        # Como es una estimación se optó por el punto medio del
        # ancho dx del rectángulo
        integral += eval(funcion+"("+str(x+dx/2)+")")*float(dx)

        # Se aumenta el contador para avanzar a lo largo del
        # intervalo.
        x += dx

    # Al final se agrega la instrucción "return" que permite
       devolver
    # el valor total calculado.
    return integral

# Estimamos numéricamente, a través de la integral de Riemann
# el valor de la integral de coseno en el intervalo dado.
numerico = calcularIntegralRiemann("math.cos",0.01,0,math.pi/2)

# Estimamos analíticamente el valor de la integral de coseno,
# que equilvale al seno, en el intervalo dado.
analitico = math.sin(math.pi/2) - math.sin(0)

# Mostramos los resultados
print "La integral numérica es",numerico
print "La integral analítica es",analitico
```

Para utilizar el valor calculado con la función **`calcularIntegralRiemann`** basta con crear una variable que almacene el valor, en este caso **`numerico`** . Luego puede utilizarse dicha variable para distintos fines, como mostrarlo a pantalla.

Es importante destacar que este programa podría utilizarse para estimar el valor de la integral de cualquier función matemática continua en el intervalo de integración.

Actividad

Utilizando el programa anterior cree una función que calcule el error porcentual entre la integral numérica y la integral analítica.

Para esto considere que el error porcentual se calcular como:

```
|valor estimativo - valor real| * 100 / valor real
```

Ahora que ya sabemos crear funciones y hemos visto algunos códigos, podemos ver que la ubicación en el programa de las funciones no es arbitraria. Esto debido a que los programas poseen una estructura que ayuda a la legibilidad y que estos puedan ser leídos y entendidos fácilmente por otros programadores.

En muchos aspectos, el intérprete de Python no requiere un programa muy estructurado, esto significa que es posible que un programa que no esté correctamente estructurado funcione, sin embargo, un programa que diferencia claramente secciones de código distintas tiene mejor legibilidad. Tratemos siempre de mantener el siguiente esquema de organización:

Encabezado	Descripción general, Autor(es), Fecha/versión	En esta parte del código se identifica el programa, indicando cuál es objetivo del programa, quién es el autor, y que versión del código es.
Bloque de definiciones	Importación de módulos	Las primeras líneas de código del programa indican,los módulos que Python requiere para que el código funcione
	Definición de constantes	En este punto se definen las constantes, que son,los valores que no se alterarán durante todo el código
	Definición de funciones	En esta parte se indican las funciones que hemos,creado para nuestro programa.
Bloque principal	Entrada de datos	Corresponden a las entradas, hasta el momento hemos utilizado únicamente `input()` y `raw_input()` para ello, sin embargo, más adelante pueden incluirse funciones que realicen dichos procesos.
	Procesamiento	Corresponden a las transformaciones, cálculos o,procesos que se hacen sobre las entradas.
	Entrega de los resultados (salida)	Corresponde a la parte del código encargada de,hacer entrega de los resultados al usuario.

Si vuelves a mirar los códigos de programa utilizados hasta el momento, todos ellos cumplen con esa estructura.

Ejercicio Propuesto

Elabore un programa en Python que sea capaz de calcular la distancia entre dos puntos dados en coordenadas cartesianas. Para esto recuerde anotar antes las ideas que organizan su programa y motivan el código.

Recuerde crear al menos una función en su implementación.