

Subprogramas

[7.1] ¿Cómo estudiar este tema?

[7.2] Introducción a los subprogramas

[7.3] Funciones y procedimientos

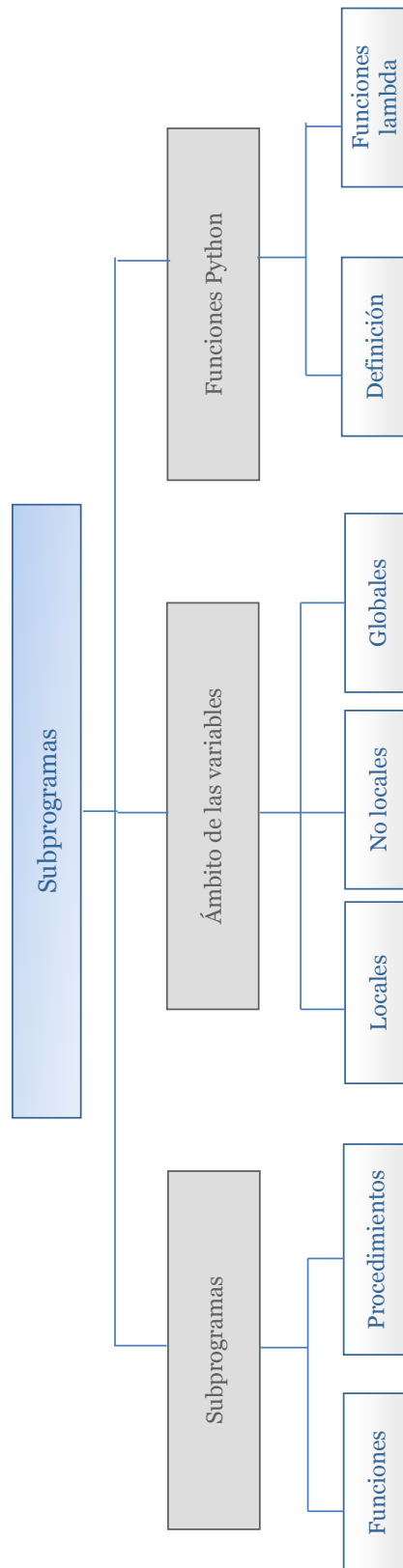
[7.4] Ámbito y visibilidad

[7.5] Funciones en Python

7

TEMA

Esquema



Ideas clave

7.1. ¿Cómo estudiar este tema?

Para estudiar este tema deberás comprender las **Ideas clave** expuestas en este documento, que se complementan con lecturas y otros recursos para que puedas ampliar los conocimientos sobre el mismo.

En este tema se introducen los **subprogramas**. Por tanto, es necesario conocer

- » **Funciones y procedimientos.**
- » **Ámbito** de las variables de un subprograma.
- » **Funciones en Python.**

7.2. Introducción a los subprogramas

Como ya se vio, en el «Tema 2», la descomposición modular de un problema en problemas más sencillos permite simplificar su resolución. Es lo que se conoce como la técnica de **divide y vencerás**.

Supongamos que se quiere hacer un programa que dibuje una figura, que puede ser una círculo, un triángulo o dos líneas que se cortan (Figura 6.1). El programa principal, se divide en subprogramas más sencillos, que reciben el nombre de subprogramas, lo que facilita la comprensión, la ejecución y el mantenimiento del programa. Cada subprograma puede contener a su vez otros subprogramas más sencillos.

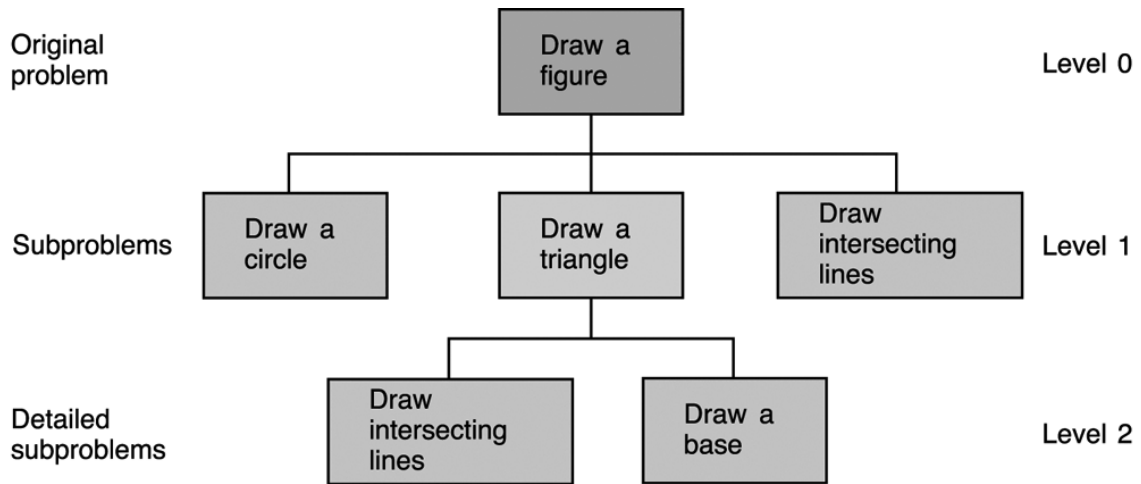


Figura 6.1: Diseño descendente de un programa. Fuente: <http://www.cis.temple.edu/>

Además, en muchas ocasiones, los subprogramas pueden diseñarse de manera independiente.

De esta manera, se tiene un **programa principal que va llamando o invocando a distintos subprogramas**, que a su vez pueden llamar a otros subprogramas contenidos en ellos (Figura 6.2). **Cuando los subprogramas realizan sus tareas, devuelven el control al programa principal o al subprograma que los contiene.**

Otra ventaja que tiene esta forma de trabajar es que se puede reutilizar el código. Por ejemplo, en la Figura 6.1 el subprograma que dibuja líneas secantes se emplea en los niveles 1 y 2 y no es necesario escribir todo el código de nuevo, basta con llamar al subprograma “Draw intersecting lines” en dos sitios distintos.

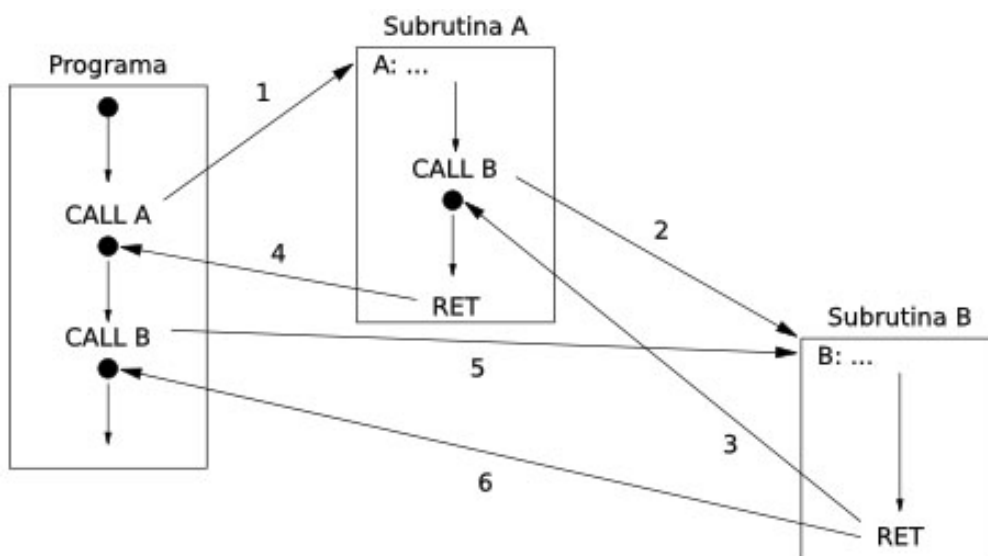


Figura 6.2: Invocación anidada de subprogramas. Fuente: <http://ocw.uc3m.es/>

No debe verse esta sucesión de flechas (llamadas y devoluciones) como algo exclusivo de la programación, sino como una representación de la manera en que se resuelven muchos problemas, por ejemplo, en una cadena de montaje, la cinta transportadora sería el programa principal y las distintas manipulaciones al producto, llamadas a subprogramas, que, una vez finalizadas, devuelven el control al programa principal.

7.3. Funciones y procedimientos

Funciones y procedimientos son los dos tipos de **subprogramas** aceptados por la mayoría de lenguajes de programación. De hecho, en función del lenguaje utilizado, pueden variar las definiciones de funciones y procedimientos.

En Python no existe la distinción entre funciones y procedimientos, que se llaman funciones de forma genérica, por lo que todos los algoritmos que se vean en este tema se implementarán del mismo modo (esto se verá en detalle al final del tema).

Para ilustrar este apartado, vamos a considerar un programa que tome una lista que contiene la estatura de un grupo de niños, devuelva la estatura media e imprima la lista ordenada.

Aunque las listas se explican en detalle en el «tema 8», para comprender el código de los siguientes ejemplos, basta con recordar la definición de lista que se dio en el «tema 3» y saber que, en Python

- » Para definir una lista, basta con escribir sus elementos entre corchetes, separados por comas: `l = ["a", 2, "g", "f"]`, define la variable `l` como una lista que contiene los elementos `"a"`, `2`, `"g"`, `"f"`.
- » La lista vacía se define con corchetes que no encierran nada, `[]`.
- » `len()` devuelve la longitud de una lista, en el ejemplo anterior, `len(l) = 4`.

Funciones

Las **funciones** son operaciones que **toman una serie de valores de entrada** o argumentos y **devuelven un determinado valor** como resultado.

Existen funciones ya incorporadas en el lenguaje y funciones definidas por el usuario. Por ejemplo, son funciones

- » El máximo de una serie de valores, x_1, x_2, \dots, x_n .
- » La raíz cuadrada de un número.

En el primer caso, la función tiene n parámetros de entrada y en el segundo caso, solo uno, pero en ambos casos devuelve un único dato de salida. Todos los lenguajes de programación incorporan funciones para resolver los dos ejemplos anteriores.

Si se quiere hacer algo más específico, como por ejemplo, devolver la suma de una serie de elementos elevados al cubo, habrá que implementar una función que lo resuelva. En los lenguajes que hacen distinción entre funciones y procedimientos, para definir una función, hay que indicar

- » El nombre de la función.
- » Los parámetros que emplea.
- » El valor que devuelve como salida.

En el ejemplo de la lista con las estaturas, la parte que habría que implementar con una función es la que calcula la estatura media. Supongamos que dicha función no estuviera implementada ya en nuestro lenguaje, una implementación en pseudocódigo sería

```

alg media
  var
    L:lista
    m:real
  inicio
    leer(L)
    N ← longitud(L)
    m ← 0
    desde i ← 1 hasta N hacer
      m ← m + L[i]
    fdesde
    m ← m/N
  falg

```

Nótese que en este pseudocódigo se ha utilizado la función `longitud()`, que en Python se implementa como `len()` y así evitamos introducir como variable la longitud de la lista, como se venía haciendo hasta ahora.

Procedimientos

Los **procedimientos** son subprogramas que permiten realizar más tipos de tareas que las funciones, ya que **no tienen que devolver un único valor**, sino que pueden devolver varios o ninguno.

Ejemplos de procedimientos son:

- » Un subprograma que ordene una lista.
- » Un subprograma que lea un nombre introducido por teclado e imprima ('Hola', nombre)

Tal y como ocurre con las funciones, los procedimientos pueden ser proporcionados por el lenguaje de programación o escritos por el usuario.

En todo procedimiento habrá que indicar siempre:

- » El nombre de la función.
- » Los parámetros que emplea.

El valor de salida no se indica porque de hecho no es necesario que exista ninguno.

En el ejemplo de las estaturas, el subprograma sería aquel que imprime la lista ordenada. Como ya vimos en el tema 4 un algoritmo de ordenación, ordenar, vamos a suponer que ya está implementado y invocamos como subprocedimiento.

```
alg imprimir
var
    L: lista
inicio
    leer(L)
    escribir(ordenar(L))
falg
```

Por tanto, la representación del programa de ejemplo sería (Figura 6.3)

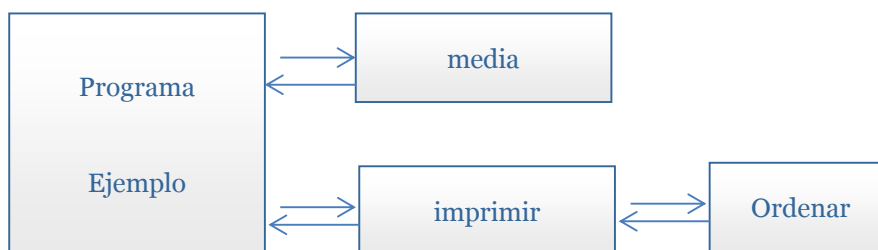


Figura 6.3: Invocación de subprogramas en el programa ejemplo

7.4. Ámbito y visibilidad

Cuando se escribe un subprograma es muy habitual necesitar variables auxiliares en el algoritmo que contengan los cálculos intermedios. Entonces, al utilizar varios subprogramas en un mismo algoritmo, podría ocurrir que algunas de esas variables auxiliares tuvieran el mismo nombre y como consecuencia se produjeran resultados imprevistos.

Para evitar estos problemas, todos los lenguajes de programación limitan el ámbito de las variables, distinguiendo entre dos tipos, globales y locales (Figura 6.4).

Las **variables globales** son aquellas **accesibles por cualquier parte del programa**, es decir, por el programa principal y por cualquier subprograma en él.

Las **variables locales** son aquellas cuyo **ámbito se restringe al subprograma** en que se ha creado.

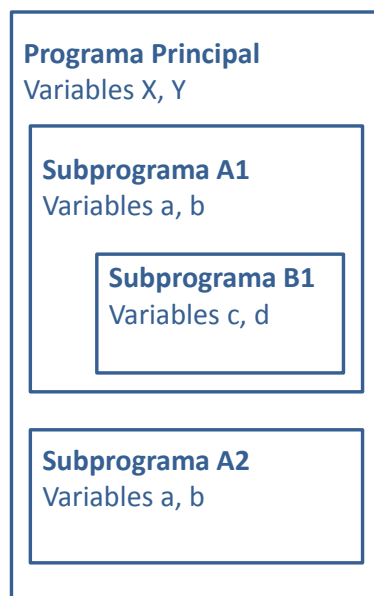


Figura 6.4. Representación del ámbito de variables en un programa.

Las variables X e Y son globales: están definidas en el programa principal y, por tanto, son accesibles para todos los subprogramas. Las variables a y b están definidas en los subprogramas A1 y A2, por tanto son locales. Como A1 no puede acceder a las variables de A2 y viceversa, no se produce ningún conflicto entre los nombres. El programa principal no puede acceder a las variables a y b de ningún subprograma, pero el subprograma B1 sí puede acceder a las variables definidas en A1. En cambio, c y d solo son accesibles para B1.

La Figura 6.4 representa el marco general en el ámbito de variables. Python también responde a este marco, pero con algunas particularidades. Python añade un nuevo ámbito, distinguiendo así entre tres tipos de variables: locales, no locales y globales.

- » **Variables locales:** pertenecen al ámbito del subprograma y son accesibles desde niveles inferiores.
- » **Variables no locales:** pertenecen a un ámbito superior al de la subrutina, pero sin ser globales. Es decir, plantean una situación intermedia entre las variables locales y las globales. Si, en la Figura 6.4, las variables c y d del subprograma B1 fueran no locales, c y d serían accesibles desde A1, pero no desde el programa principal.
- » **Variables globales:** pertenecen al ámbito del programa principal, siendo así accesibles desde cualquier nivel.

En muchos lenguajes de programación es necesario declarar las variables que van a utilizarse en un programa. En la declaración también se explicita el ámbito de la variable. Pero en Python las variables no se declaran, por lo que el ámbito al que pertenece viene implícito.

- » Si en un subprograma se definen variables, **Python por defecto va a considerarlas variables locales**. Por ejemplo, el código

```
def f():
    s = "Variable local"
    print (s)
s = "Variable global"
f()
print (s)
```

devuelve

```
Variable local
Variable global
```

En primer lugar, vamos a fijarnos en la sintaxis. La primera línea, `def f()`, indica que estamos definiendo una función `f` que no precisa argumentos de entrada. La segunda línea define una variable `s`, que, al estar dentro de una función, en principio va a ser local y en la tercera y última línea de la función se pide que se imprima `s` por pantalla. Sabemos que es la última línea de `f` por la indentación. A continuación, se define una variable global `s`, se ejecuta `f()` y se pide de nuevo imprimir `s`.

A pesar de que hay dos variables `s` definidas (una local y una global), el ámbito por defecto es local, por eso cuando se ejecuta `f()` se muestra el mensaje `Variable local`, pero cuando se manda mostrar `s` por pantalla en el cuerpo principal, el mensaje cambia, ya que se está considerando la variable global.

De hecho, si se ejecuta el código

```
def f():
    s = "Variable local"
    print (s)
f()
print (s)
```

Se produce un error, ya que `s` es una variable local a la que el programa principal no tiene acceso.

- » Una función utilizará una variable global cuando no haya ninguna definida en la función con el mismo nombre. Por ejemplo, al ejecutar

```
def f():
    print (s)

s = "variable global"
f()
```

Se obtiene

```
variable global
```

También asume que la variable es local en el código

```
def f():
    print(s)
    s = "Variable local."
    print (s)

s = "Variable global"
f()
print (s)
```

Que devuelve el error

```
UnboundLocalError: local variable 's' referenced before assignment
```

¿Qué ha pasado para que en este caso hayamos obtenido un mensaje de error? Como dentro de la función lo primero que hay es una llamada a `s`, que aún no está definida localmente, Python podría asumir que se trata de una variable global. Pero en la siguiente línea se encuentra una asignación que, al estar dentro de una función, se corresponde con una variable local. Por eso se muestra un mensaje de error.

- » Cuando se quiere especificar que una variable que se define dentro de una función es global o no local, puede indicarse con `global` o `nonlocal`, respectivamente. Por ejemplo, si modificamos el código anterior

```
def f():
    global s
    print(s)
    s = "Variable local."
    print (s)

s = "Variable global"
f()
print (s)
```

Devuelve

```
Variable global
Variable local.
Variable local.
```

Ya no hay conflicto entre variables porque, aunque se hace una asignación a `s` dentro de `f`, se ha especificado que se trata de una variable local.

Veamos por último un ejemplo con variables locales, no locales y globales

```
def prueba_ambitos():
    def var_local():
        s = "variable local"
    def var_nonlocal():
        nonlocal s
        s = "variable no local"
    def var_global():
        global s
        s = "variable global"
    s = "variable 1"
    var_local()
    print("Despues de la asignación local:", s)
    var_nonlocal()
    print("Despues de la asignación no local:", s)
    var_global()
    print("Despues de la asignación global:", s)

prueba_ambitos()
print("En el programa principal:", s)
```

Produce como resultado

```

Despues de la asignación local: variable 1
Despues de la asignación no local: variable no local
Despues de la asignación global: variable no local
En el programa principal: variable global

```

Las herramientas que explican este resultado ya se han expuesto, por lo que se deja al alumno la explicación del resultado obtenido.

7.5. Efectos laterales

Como las variables globales pueden utilizarse no solo en el programa principal, sino en cualquier subprograma, es posible que uno de estos subprogramas modifique alguna variable global. Esto es lo que se conoce como **efecto lateral**.

Los efectos laterales, en general, deben evitarse, ya que reducen la independencia entre los distintos módulos del programa y puede producir errores muy difíciles de detectar y depurar. Por tanto, siempre que sea posible, deben utilizarse variables locales en Python y modificar las globales solo dentro del programa principal.

7.6. Funciones en Python

En Python no se hace la distinción entre funciones y procedimientos, todos los subprogramas son funciones. De hecho, aunque se implemente un subprograma que no retorne ningún valor, Python siempre devolverá `None`.

Hay que tener en cuenta que Python dispone de gran cantidad de funciones integradas. Si a esto le sumamos las librerías estándar y los módulos creados por terceros, resulta que ya hay implementadas cientos de funciones. Por esto, antes de lanzarse a implementar código nuevo, merece la pena comprobar si ya hay algo similar implementado.

La sintaxis general para crear una función es:

```

def nombreFunción(parámetros):
    <acciones>

```

De nuevo, no hay ninguna palabra o símbolo que indique el final de la función, sino que se marca con la indentación. Esto obliga al programador a indentar correctamente el código y, junto con la sintaxis clara de Python, hace que los programas sean muy fácilmente entendibles por un ser humano.

Los parámetros son opcionales. Si no se quiere poner ninguno, simplemente se escribirá

```
def nombreFunción():
    <acciones>
```

Y si hay más de uno pueden escribirse en secuencia, separándolos por comas

```
def nombreFunción(parámetro1, parámetro2, ..., parámetro n):
    <acciones>
```

Por ejemplo, si se quiere hacer una función que calcule el perímetro y el área de un rectángulo a partir de sus lados

```
def perimetro_area(a, b):
    perimetro = 2 * (a + b)
    area = a * b
    return(perimetro, area)
```

Para ejecutar esta función hay que pasar todos los argumentos, por ejemplo `perimetro_area(3, 4)`. Cada parámetro se inicializa con los valores `a` y `b` que se pasen como argumento, así, en el ejemplo anterior, `a` se ha inicializado con 3 y `b` con 4.

La sentencia `return` indica qué valores se devuelven tras ejecutar la función. Los valores pueden ser un número, una cadena, una tupla con varios valores o `None`, en caso de que no haya sentencia `return`.

Es posible definir valores por defecto en los parámetros de una función. Por ejemplo,

```
def perimetro_area(a = 1, b = 2):
    perimetro = 2 * (a + b)
    area = a * b
    return(perimetro, area)
```

Calculará el perímetro y el área de un rectángulo de lados 1 y 2 si se invoca a la función sin parámetros `perimetro_area()`. No es necesario poner valores por defecto en todos los parámetros, lo que sí debe hacerse es poner en último lugar los parámetros con valores por defecto. Por tanto, una función que comience con algo como `def perimetro_area(a = 1, b):`, producirá un error.

Supongamos que tenemos una función que, dada una cadena, decide si es larga a partir del número de caracteres y en caso de que lo sea, devuelve una versión más corta, que llevará una marca indicando que se ha abreviado. Esta función podría implementarse

```
def cadena_corta(cadena, longitud = 10, marca = "[ ]"):
    if (len(cadena) > longitud):
        maximo = longitud - len(marca)
        cadena = cadena[:maximo] + marca
    return(cadena)
```

Es posible invocar cualquier función introduciendo los argumentos en el orden que se quiera. Por ejemplo

```
cadena_corta("Ejemplo") devuelve
'Ejemplo'
```

```
cadena_corta(longitud = 5, cadena = "Ejemplo") devuelve
'Eje[]'
```

```
cadena_corta("Ejemplo", marca = "*-*", longitud = 5) devuelve
'Ej*-*'
```

```
cadena_corta("Ejemplo", 5, "*-*") devuelve
'Ej*-*'
```

Los parámetros `longitud` y `marca` tienen valores por defecto, por lo que no es necesario introducir los argumentos correspondientes, como se puede ver en la primera llamada a la función. En la segunda y la tercera llamadas, como se utilizan argumentos de palabra clave, se pueden poner en cualquier orden. En la cuarta se introducen todos los argumentos por orden, por lo que no es necesaria la palabra clave.

Nótese que en todos los ejemplos se ha introducido el argumento de cadena. Este argumento es obligatorio ya que, al no tener valores por defecto, es imposible que Python pueda saber qué valor utilizar si no se introduce ninguno. En cambio, los valores `longitud` y `marca` son opcionales.

Los argumentos opcionales resultan especialmente útiles en funciones que tienen muchos argumentos de entrada: no es necesario especificarlos todos cada vez que se invoca a la función, lo que agiliza mucho la lectura y escritura del código.

Funciones lambda

Una **función lambda** es una función que se define en una sola línea, no admite estructuras repetitivas ni selectivas y, como siempre devuelven un valor, no pueden incluir una sentencia `return`.

Dicho de manera más formal, el operador lambda permite crear **funciones anónimas**, esto es, funciones sin nombre. Estas funciones se crean donde se necesitan se ejecutan y no se almacenan, por lo que son especialmente útiles para funciones cortas que solo se usarán una vez.

La sintaxis de una función lambda es

```
lambda parámetros: expresión
```

Por ejemplo

```
prod_menos2 = lambda x, y: (x - 2) * (y - 2)
prod_menos2(5, 7)
15
```

Toma los dos argumentos de entrada, les resta 2 y a continuación los multiplica.

```
area = lambda base, altura: base * altura / 2
area(4, 8)
16.0
```

Calcula el área de un triángulo, cuando se introducen la base y la altura.

Estas funciones también podrían escribirse como

```
def prod_menos2 (x, y):
    prod_menos2 = (x - 2) * (y - 2)
    return(prod_menos2)

def area (base, altura):
    area = (base) * (altura) / 2
    return(area)
```

La ventaja que tienen es que reducen líneas de código, ya que se escriben en una sola línea. No obstante, es preferible definir una función (aunque sea muy sencilla) si se va a emplear muy a menudo en el código que utilizar su equivalente lambda.

Nombres de funciones y variables

Las normas básicas que se deben seguir a la hora de asignar nombres ya se han descrito. No obstante, es conveniente ser un poco más metódico en la elección de los nombres para facilitar la lectura del código. Unas reglas recomendables son:

- » Elegir nombre de variables que describan los datos que guardan. Aunque en principio, para las variables auxiliares de una función, resulte más cómodo utilizar nombres como 'a' que nombres como 'intereses_anuales', cuando al cabo de un tiempo un tercero revise la función (o incluso nosotros mismos), la comprenderá mucho antes si se han utilizado nombres significativos para las variables.

No hay que llevar esta regla tampoco al extremo: el índice de un bucle no hace falta que se llame 'índice', las variables 'i' o 'j' son adecuadas. Algo análogo ocurre, por ejemplo, para referirnos al radio de una circunferencia: 'r' o 'R' son nombres que pueden emplearse tranquilamente.

- » Ser sistemático con el esquema de nombres escogido. Por ejemplo se puede decidir que todas las funciones tendrán un nombre en minúsculas o minúsculas con barras bajas o utilizar mayúsculas para las constantes (recordemos además que, en Python, no se distinguen formalmente las variables de las constantes, por lo que es una buena práctica para recordarle al programador que no se deben modificar).

- » Evitar abreviar todos los nombres, ya que a la larga, dificultan la comprensión de la palabra que se abrevia.
- » Asignar nombres a las funciones que indiquen qué hacen o qué devuelven de la forma más clara posible. Por ejemplo, se tiene una función que calcula el área de un triángulo en función de su base y su altura. De las siguientes propuestas

```
area(a, b)  
area_triangulo(base, altura)
```

La segunda describe mucho mejor qué se hace y cuáles son los argumentos que se necesitan.

Lo + recomendado

No dejes de leer...

Tutorial del módulo Turtle de Python

En este enlace puede verse un tutorial para utilizar el módulo Turtle de Python, para dibujar.

Accede al documento a través del aula virtual o desde la siguiente dirección web:

http://www.openbookproject.net/thinkcs/archive/python/thinkcspyesp3e_abandonado/cap03.html

Este segundo que está en inglés es más completo.

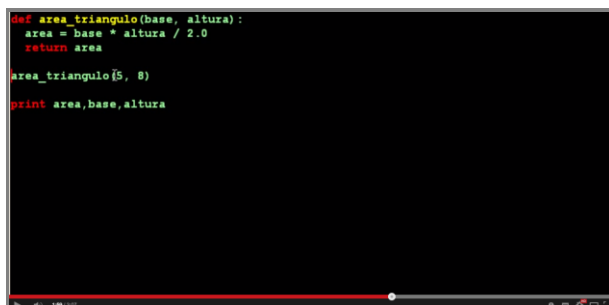
Accede al documento a través del aula virtual o desde la siguiente dirección web:

<http://openbookproject.net/thinkcs/python/english3e/functions.html>

No dejes de ver...

Variables locales y globales en Python

En este enlace se puede ver un ejemplo del ámbito de variables en Python



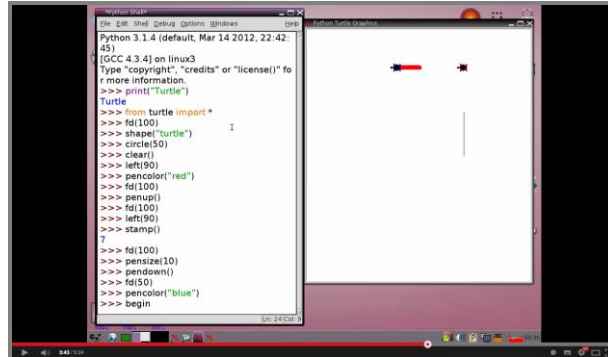
```
def area_triángulo(base, altura):  
    area = base * altura / 2.0  
    return area  
  
area_triángulo(5, 8)  
print area, base, altura
```

Accede al vídeo a través del aula virtual o desde la siguiente dirección web:

<https://www.youtube.com/watch?v=oQbaKQZ88jQ>

Tutorial del módulo Turtle de Python

En este enlace, se puede ver un video tutorial para el uso de Turtle.



Accede al vídeo a través del aula virtual o desde la siguiente dirección web:

<https://www.youtube.com/watch?v=jozHKiSo5Z4>

Curso de Python 3.x. Python orientado a objetos

Iniciación a la programación orientada a objetos en Python.



Accede al vídeo a través del aula virtual o desde la siguiente dirección web:

<https://www.youtube.com/watch?v=YfevwIq3Jts>

+ Información

A fondo

Built in Functions

Descripción de las funciones predefinidas en Python.

Accede al documento a través del aula virtual o desde la siguiente dirección web:

<https://docs.python.org/3/library/functions.html>

Procedures and Functions

Definición y uso de funciones y procedimientos.

Accede al documento a través del aula virtual o desde la siguiente dirección web:

<http://flint.cs.yale.edu/cs422/doc/art-of-asm/pdf/CH11.PDF>

Functions

Ejemplos de código Python en distintas funciones.

Accede al documento a través del aula virtual o desde la siguiente dirección web:

<http://www.sthurlow.com/python/lesson05/>

Enlaces relacionados

Python

Python es un lenguaje de interpretado, gratuito y de código abierto.



Accede a la página web a través del aula virtual o desde la siguiente dirección:

<https://www.python.org/>

Tutoriales de Python: <http://docs.python.org.ar/tutorial/pdfs/TutorialPython3.pdf>

Bibliografía

Joyanes, L. (2008). *Fundamentos de la programación. Algoritmos y Estructura de Datos* (4ª Ed). Madrid: McGraw-Hill.

Summerfield, M. (2009). *Programación en Python 3*. Madrid: Pearson Educación.

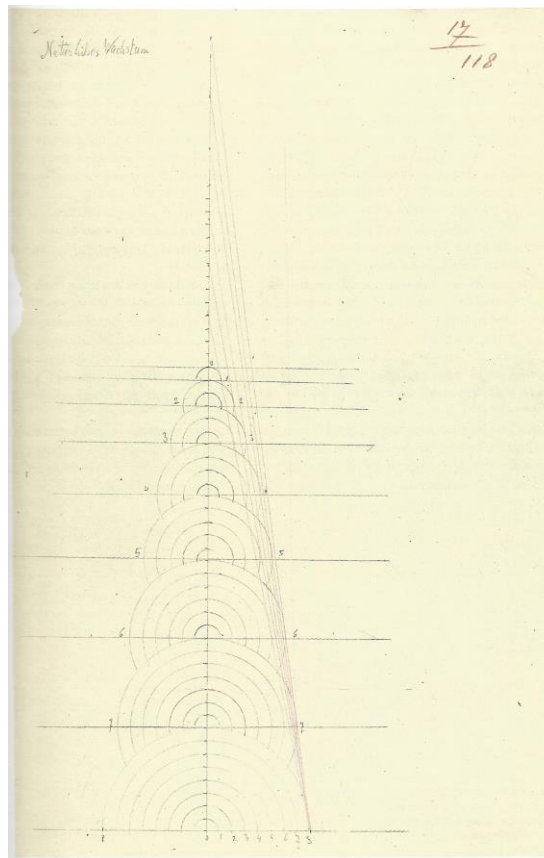
Valls, J. M. & Camacho, C. (2004). *Programación estructurada y algoritmos en PASCAL*. Madrid: Pearson Educación.

Actividades

Trabajo: Copiar una imagen

Metodología

- » Realizar una copia de la imagen que se adjunta utilizando en módulo de Python Turtle.



Paul Klee, Progresiones B II, 19/87

Criterios de evaluación

- » Se evaluará el diseño y la claridad del programa así como el parecido de la figura generada con el modelo.

Extensión máxima: 8 páginas 1 página de portada, 1 página de índice, 1 página para explicar las estructura del programa, 5 páginas para escribir el código de las funciones, como máximo), fuente Georgia 11 e interlineado 1,5.

Test

1. Los subprogramas:

- A. Dividen un problema en subproblemas más sencillos.
- B. Permiten reutilizar el código.
- C. A y B son ciertas.

2. Las funciones:

- A. Devuelven siempre varios valores como resultado.
- B. Todos los lenguajes de programación incorporan algunas ya predefinidas.
- C. Siempre incluyen estructuras selectivas.

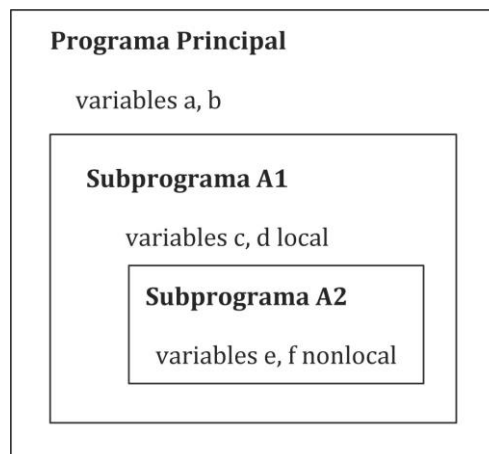
3. Un procedimiento:

- A. No tiene por qué devolver ningún valor.
- B. No puede devolver ningún valor.
- C. A y B son falsas.

4. Cuando dos funciones distintas en un mismo subprograma tienen definidas variables locales con el mismo nombre:

- A. Los valores de las variables se sobrescriben.
- B. No hay conflicto entre los nombres.
- C. Es recomendable cambiar los nombres en una de las funciones, para que no coincidan.

5. En Python, en el subprograma A1 de la siguiente figura son visibles las variables:



- A. a, b, c, d, e, f
- B. c, d, e, f
- C. c, d

6. Que las variables globales se modifiquen localmente se conoce como:

- A. Ámbito.
- B. Visibilidad.
- C. Efectos laterales.

7. En Python:

- A. Solo las funciones con sentencia `return` devuelven un valor.
- B. Todas las funciones retornan un valor.
- C. Las funciones no pueden retornar más que un entero o una cadena.

8. En Python se verifica que:

- A. Se establece una marca para designar el fin de la definición de una función.
- B. El final de una función se marca con la indentación.
- C. A y B son correctas.

9. Cuando se especifica un valor por defecto:

- A. No deben introducirse valores para ese parámetro.
- B. El resto de argumentos deben tener también valores por defecto.
- C. No es necesario introducir valores para ese parámetro.

10. Una función lambda:

- A. No admite sentencia `return` porque no devuelve nada.
- B. Sirve para implementar funciones que no pueden implementarse de otro modo.
- C. Es una función anónima.