SEAS
CAMPUS SEAS

4 UNIDAD DIDACTICA

Python

4. Procedimientos, funciones y diseño descendente

ÍNDICE

OBJETIVOS 127
INTRODUCCIÓN 128
4.1. Escritura de procedimientos 129
4.2. Llamadas a procedimiento 131
4.3. Clases de parámetros 133
4.4. Búsqueda secuencial 138
4.5. Funciones 139
4.6. Ámbito de los objetos informáticos 141
4.7. Ventajas de la localidad 143
4.8. Generación de números aleatorios 144
4.9. Anidamiento de ámbitos 146
4.10. Efectos colaterales 148
4.11. Recursividad 150
4.12. Captura y tratamiento de excepciones 152
4.13. Lanzamiento de una excepción 155
4.14. Diseño descendente 156
RESUMEN 157

OBJETIVOS

El objetivo fundamental de esta unidad es aprender a enfocar la resolución de problemas más o menos complejos aplicando diseño descendente. Para ello, se procede a descomponer los problemas en otros de menor entidad, que se pueden resolver con relativa sencillez y que posteriormente se pueden aplicar para la resolución de otros problemas distintos del que se pretende resolver. A su vez, estos problemas de menor entidad los pueden resolver equipos de programadores distintos sin que exista ningún problema de comunicación entre ellos.

Para ello, estos problemas de menor entidad se resuelven mediante el uso de procedimientos o funciones. Su uso, permite obtener un código limpio y legible, y facilita considerablemente la programación.

Resumiendo, los objetivos de esta unidad son:

- Aprender a escribir procedimientos y funciones en Python.
- Profundizar en el paso de parámetros.
- Comprender las variables locales y globales.
- Utilizar procedimientos y funciones en programas de mayor entidad mediante diseño descendente.
- Conocer el uso de las excepciones a nivel básico.

INTRODUCCIÓN

En esta unidad vamos a desarrollar las herramientas de programación que nos permitirán resolver problemas complejos mediante su descomposición en otros más simples, como los expuestos en la unidad anterior. Es la base de la metodología de diseño *descendente* que se basa en dividir un programa complejo en otros más simples para así facilitar su diseño.

La programación basada en procedimientos y funciones permite esa descomposición, mejora la legibilidad, la depuración y el mantenimiento de los programas.

Hay que destacar que el mayor logro de esta técnica de trabajo es la transportabilidad, es decir, los procedimientos y las funciones se deben construir de forma que sean transportables a cualquier otro programa que realice la misma función.

En su forma más simple, se puede decir que un procedimiento es una instrucción virtual que sustituye a una secuencia de instrucciones que tenga un cierto sentido en sí misma, a la que se da cierto nombre. Por tanto, hay que destacar:

- Equivalen a una instrucción que sustituye a una secuencia de instrucciones.
- Deben tener sentido en sí mismas.
- Se le da un nombre.

Por ejemplo, se podría concebir una función que sirviese para generar un número aleatorio.

Dado que una instrucción de este tipo se puede usar en multitud de programas y aplicaciones, en lugar de escribir la secuencia de instrucciones que nos permiten obtener una secuencia de números aleatorios cada vez que se necesita en un programa, se definirá una función a la que daremos nombre (por ejemplo, aleatorio) y se usará esta función en todos los programas en los que se necesite. Para ello, bastará con escribir el nombre de la función en la forma:

```
aleatorio()
```

Tras ejecutarse la función *aleatorio* se dispondrá de un número aleatorio generado por la función.

Evidentemente, la declaración de la secuencia de instrucciones que sirven para generar el número aleatorio se realizará en el seno de la función en la forma que se verá más adelante (mediante su declaración). A su vez, aleatorio() se dice que es una llamada a la función aleatoria.

A modo de ejemplos, en la tercera Unidad se vieron acciones tales como: realizar un cambio de moneda, calcular el factorial de un número, etc. Tales acciones se pueden ver como instrucciones virtuales, es decir, como procedimientos o funciones que se implementan independientemente del programa y posiblemente por distintos programadores que no tienen apenas comunicación entre sí.

En esta unidad se van a sentar las bases del diseño descendente basado en la utilización de procedimientos y funciones, y se estudiará la forma de escribirlos y de utilizarlos en programas de cierta complejidad.

4.1. Escritura de procedimientos

En este apartado vamos a ver en primer lugar la idea conceptual acerca de los procedimientos y las funciones, los parámetros que se les pasan y la forma de declararlos.

En la introducción ya se ha comentado cual es esta idea. Se trata de instrucciones virtuales que sustituyen a una secuencia de instrucciones que tenga un cierto sentido en sí misma, a la que se da cierto nombre. Para poder operar pueden hacer uso de datos que se les pasan por medio de parámetros. También pueden devolver resultados.

Parámetros de los procedimientos

Algunos procedimientos únicamente generan resultados sin necesidad de pasarles ningún valor con el que operar; por ejemplo, "generar un número aleatorio" no necesita de ningún valor previo, únicamente genera un resultado.

Sin embargo, otros sí que necesitan unos valores a partir de los cuales van a generar los resultados; por ejemplo, un hipotético "dibujar un punto en la posición (X,Y)", necesita conocer la posición en la que se va a dibujar el punto. En este caso no se obtendrá ningún valor como resultado, ya que únicamente dibujará un punto en la pantalla. Hablamos de procedimientos.

Otro ejemplo puede ser calcular el factorial de un número. En este caso se necesita pasarle el número cuyo factorial queremos calcular y se obtiene como resultado el valor de dicho factorial. Hablamos de funciones.

Por tanto, a las funciones (al igual que a una función matemática) y a los procedimientos se les pueden pasar valores con los que puede operar y pueden generar resultados. Dichos valores se le pasan por medio de parámetros.

Estas ideas nos permiten definir un *procedimiento o una función* como: "un algoritmo normalmente con parámetros (puede no haberlos) que define una instrucción virtual".

Para poder llamar a un procedimiento o función previamente habrá que declararlos, es decir, habrá que escribirlos. La declaración consiste en una cabecera y un cuerpo del procedimiento o la función y debe hacerse antes de su llamada.

- La cabecera del procedimiento o función ocupa una línea de código que contiene:
  - La palabra reservada **def** indicando que se trata de un procedimiento o función,
  - el identificador (nombre) del procedimiento, y
  - tras el identificador, entre paréntesis se escriben los parámetros (si los hay). Dichos parámetros se conocen con el nombre de parámetros formales y van escritos separados por comas.
  - Termina en ":"

- El cuerpo, que contiene el código, está compuesto por una secuencia de instrucciones que se van a ejecutar cuando se produzca una llamada al procedimiento o función.

- En el caso de una función hay que añadir en el cuerpo de la función una cláusula **return** seguida de una variable. La función devuelve el valor de la variable que sigue a return.

De acuerdo con esto, un procedimiento para escribir la famosa frase “Hola Mundo”, quedaría en la forma siguiente:

```
def frase (): ← Cabecera
    print(“Hola Mundo”) ← Cuerpo
```

Se trata de un procedimiento sin parámetros que simplemente escribe una frase en la pantalla y que no tiene sentido como procedimiento. Únicamente es un ejemplo sencillo.

4.2. Llamadas a procedimiento

En la figura 4.1 se muestra el código de un programa llamado “primero” en el que figuran dos llamadas a un procedimiento “segundo”. Diremos que el procedimiento “primero” llama al procedimiento “segundo”. Una llamada a un procedimiento o función activa al procedimiento (o la función). Para que ello ocurra, dentro del código de primero tendrá que incluirse una instrucción de llamada a segundo. Esta instrucción provocará la ejecución de segundo sobre los valores que le pasemos a través de los parámetros y nos devolverá los resultados obtenidos, si los hubiese (no es el caso).

```
def primero():
  def segundo(x,y):# segundo escribe los valores que se
  le pasan
    print(x,y)
  a, b = 1, 2       # se asignan valores a “a” y “b”
  segundo(a, b)     # se llama a segundo con esos valores
  u, v = 3, 4       # se asignan valores a “u” y “v”
  segundo(u, v)     # se llama a segundo con esos valores
primero()
```

```
1  2
3  4
```

Figura 4.1. Llamada a procedimiento.

Como se muestra en la figura 4.1, la llamada a un procedimiento se realiza escribiendo el nombre (identificador) del procedimiento seguido de las variables y/o las expresiones sobre las que deseamos que trabaje. Estas expresiones van encerradas entre paréntesis y en el mismo orden en que han sido especificadas en la declaración del procedimiento.

Una instrucción de llamada a ese procedimiento tendrá la forma: segundo (a,b)

y como consecuencia de esa llamada, se emparejan las variables, por la posición que ocupan, en la forma:

a ----- x (primer parámetro real con el primer parámetro formal)

b ----- y (segundo parámetro real con el segundo parámetro formal)

Con este protocolo hemos creado una vía de comunicación entre la instrucción de llamada y el procedimiento llamado, que facilita el intercambio de información entre ambos. Llamaremos *parámetros* a los elementos que intervienen en ese protocolo.

Los parámetros que se reciben en la instrucción de llamada se denominan *parámetros reales*. Es el caso de “a” y de “b”. Los que aparecen en la declaración del procedimiento llamado se denominan *parámetros formales*, caso de “x” e “y”.

Al emparejarse los parámetros de manera ordenada, debe coincidir el número de parámetros.

Sin embargo, no es necesario que coincidan los identificadores (los nombres) de los parámetros reales y formales. Esto posibilita, entre otras cosas, que el algoritmo anterior se pueda llamar con distintos identificadores de parámetros reales. Por ejemplo, el procedimiento anterior, se puede llamar también:

```
segundo (u,v)
```

Un procedimiento puede tener dentro varios procedimientos (procedimientos anidados) a los que puede llamar. Posteriormente se comentará.

4.3. Clases de parámetros

En el emparejamiento entre parámetros reales y formales hay aspectos que aún no hemos determinado. En particular, no hemos descrito la manera en que los cambios efectuados por la acción llamada sobre los valores de los parámetros formales, afectan a los parámetros reales de la acción principal. Tampoco hemos discutido si el programador dispone de algún tipo de control sobre ellos.

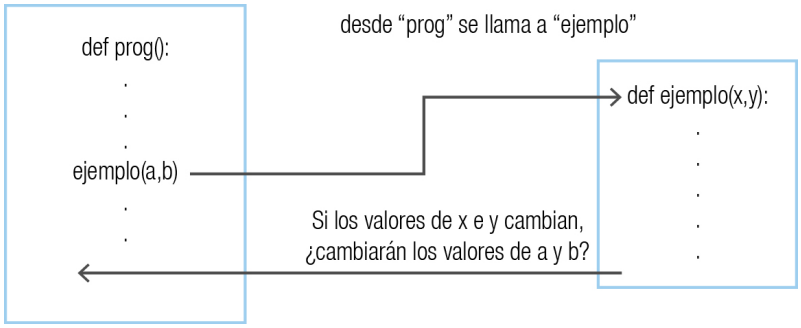

Figura 4.2. Paso de parámetros.

Este es un problema delicado, pues afecta a la protección de los datos. Para abordarlo, clasificaremos los parámetros en dos grupos acordes a los tipos de datos fundamentales existentes.

a. **Parámetros inmutables o de entrada.** Corresponden a las variables inmutables. Sabemos que los parámetros inmutables son en realidad referencias a objetos. Tales objetos no pueden cambiar de valor, aunque las referencias a ellos sí que pueden cambiar.

```
def parametros(a):
  print(a)
  a += 1

x = 2
parametros(x)
print(x)
```

```
2
2
```

Figura 4.3. Paso de parámetros inmutables.

En el código de la figura 4.3 vemos que aunque en el procedimiento se incremente en una unidad el valor de a, esto no se refleja en el valor del parámetro real x.

b. **Parámetros mutables.** Al igual que las variables mutables, pueden cambiar de valor. Así, en el código de la figura 4.4. se pasa una lista y se incrementa en una unidad el valor de cada componente de las lista modificando su valor.

```
def parametros(lst):
  n = len(lst)
  print(n)
  for i in range(n):
    lst[i]=lst[i]+1
    print(lst[i])

lista = [1, 2, 3, 4, 5, 6, 7, 8, 9]
parametros(lista)
print(lista)
```

```
9
2
3
4
5
6
7
8
9
10
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Figura 4.4. Paso de parámetros mutables.

Vemos que en el procedimiento cambia el valor de los componentes de la lista, incrementándose su valor una unidad. Pues bien, tras ejecutarse el procedimiento, el parámetro *lista* también ha cambiado el valor de sus componentes y eso se refleja en el cambio que tiene lugar en la variable *lista*.

Vamos a profundizar algo más acerca de estos parámetros.

a. Parámetros inmutables, también llamados de entrada. Son aquellos para los que los cambios en el valor del parámetro formal, no afectan al valor del parámetro real. En el ejemplo de la figura 4.3, si cambia el valor de "a" no afecta al valor de "x".

El paso de parámetros de entrada se puede describir de la forma:

- Se evalúan las expresiones de los argumentos reales usados en la llamada.
- Los valores obtenidos se copian en los parámetros formales.
- Los parámetros formales se usan como variables dentro del procedimiento. Si a estas variables se les asignan nuevos valores, no se estará modificando el parámetro real, sino solo la copia.

Por ejemplo:

```
def calculos (a,b):
  a -=1            # resta una unidad al primer parámetro
  b +=2           # suma dos unidades al segundo parámetro
  print(a,b)        # escribe los nuevos valores de los
  parámetros
  print(x1, x2)  # escribe el valor de las variables
  globales


x1, x2 = 3, 8
calculos(x1, x2)
print(x1, x2)
```

```
2 10
3 8
3 8
```

Figura 4.5. Paso de parámetros inmutables.

El uso de un parámetro que no sea de entrada comportará la posibilidad de modificar los datos de la acción que realiza la llamada.

Una consecuencia es que los parámetros reales, cuando son de entrada (inmutables), pueden ser variables o expresiones. Si figuran expresiones como parámetros, estas expresiones se evalúan antes de la activación del procedimiento.

Por ejemplo, se puede llamar al procedimiento calculos(a,b) en la forma:

```
calculos(5,7*2)
```

```
4 16
```

Figura 4.6. Llamada pasando como parámetro una expresión.

Cuando son mutables, han de ser obligatoriamente variables mutables, pues su valor se modifica al tomar por valor el que toman los parámetros formales al terminar el procedimiento.

b. Parámetros mutables. Vamos a pasar una lista, que sabemos que es mutable, por lo que puede cambiar de valor.

```
def calculos2 (a):
  a[0] = 0
  a[1] = 0
  a[2] = 0
  print(a)

x = [0, 1, 2, 3]
calculos2(x)
print(x)
```

```
[0, 0, 0, 3]
[0, 0, 0, 3]
```

Figura 4.7. Parámetros mutables.

Tras ejecutarse el procedimiento, vemos que cambia el contenido de la lista x.

También vemos que la longitud de la lista no afecta al programa, ya que el procedimiento se adapta a listas de cualquier tamaño y tipo. Por ejemplo, si se define una lista y como la de la figura 4.8, y el ejecuta el código que sigue, resulta:

```
y = ['Pepe', 12, 2, 29, 'Juan']
calculos2(y)
print(y)
```

[0, 0, 0, 29, 'Juan']

[0, 0, 0, 29, 'Juan']

Figura 4.8. Un parámetro lista se adapta a cualquier longitud.

El paso de argumentos (parámetros) por referencia se puede describir:

- Se seleccionan las variables usadas como argumentos reales.
- Se asocia cada variable con el argumento formal correspondiente.
- Se ejecutan las sentencias del procedimiento como si los argumentos formales fuesen los reales.

4.4. Búsqueda secuencial

Esta búsqueda consiste en recorrer en un sentido una lista hasta encontrar el dato buscado o hasta llegar al final. Posteriormente habrá que comprobar si ha habido éxito en la búsqueda. Dependiendo que la lista esté o no ordenada, la condición de "fin de recorrido" cambiará ligeramente. Vamos a ver en este ejemplo el caso de lista no ordenada compuesta de caracteres. Se trata de ver si el carácter almacenado en car está o no en la lista.

```
def buscar(lst, car):
    i = 0
    encontrado = False
    while i < len(lst) and not(encontrado):
        encontrado = lst[i] == car
        i += 1
  print(encontrado)

lst = ['a', 'b', 'c', 'A', 'B', 'C']
buscar(lst, 'a')
```

True

Como se muestra en el *script* que sigue, se pueden realizar búsquedas en diccionarios usando el operador in, que permite saber si una clave está en un diccionario. Por ejemplo:

```
dicc = {'c1':10, 'c2':20, 'c3':30, 'c4':40}
def buscaClave(clave):
    if clave in dicc:
      print('La clave existe')
    else:
      print('La clave no existe')
buscaClave('c2')
```

La clave existe.

4.5. Funciones

En principio, una función es un algoritmo, eventualmente parametrizado, que tiene uno o varios parámetros de salida a través de los cuales se devuelve el resultado de la función. La asignación múltiple proporciona el mecanismo que permite que haya varios parámetros de salida.

Esta particularidad permite una notación específica para la llamada a funciones. En la llamada a una función el parámetro de salida no debe aparecer en la lista de parámetros -que solo contendrá los parámetros reales de entrada- y el valor calculado por la función se referencia en la misma llamada.

Por ejemplo, si necesitamos una función que calcule el máximo común divisor de los valores “x” e “y”, y asigne este valor a la variable mcd, la llamada a la función sería:

```
mcd = maxComDiv(x,y)
```

entendiendo que maxComDiv es el nombre de la función y que la instrucción, además de evaluar la función, realiza la asignación correspondiente del resultado a la variable mcd. Se admite asignación múltiple. A su vez, como los parámetros de la función son de entrada, pueden ser expresiones. Por ejemplo:

```
def func (a,b):
  return a*b+5

z = 0
z = func(2,3*2)
print(z)
```

```
17
```

Figura 4.9. Declaración de una función.

Como ya se ha comentado, la sintaxis correspondiente a la función es la misma que la de los procedimientos pero incluyendo una o varias cláusulas *return* en el cuerpo de la función. Dicha cláusula de retorno termina la computación y pasa el resultado al punto de llamada.

Por ejemplo:

```
def min(x,y):
  if x<=y:
    return x
  else:
     return y
```

Figura 4.10. Función con dos cláusulas return.

En Python no se distinguirá entre procedimientos y funciones y se llamará a ambas funciones, por lo que podemos hablar de la existencia de dos clases de funciones: funciones propias y funciones procedimiento. Las primeras se activan (mediante una asignación de función) como un constituyente de una expresión y llega a un resultado que es un operando de la expresión. Por el contrario, las funciones procedimiento se activan mediante una llamada a procedimiento. Una función propia contiene una o varias cláusula return para generar el resultado.

4.6. Ámbito de los objetos informáticos

Cuando en un algoritmo se define un objeto informático: constante, variable, procedimiento, etc. se habla de su ámbito como el dominio en el cuál se conoce su definición y, por consiguiente, se puede utilizar el objeto.

Las reglas sobre el ámbito de un objeto pueden variar dependiendo del lenguaje considerado. Vamos a considerar que el ámbito de un objeto se situará en el algoritmo en el cuál está definido. Por ejemplo, el valor de una variable local, usada dentro de un procedimiento o función, no está definido cuando se efectúa la llamada al procedimiento e igualmente, su valor se pierde cuando el procedimiento termina.

Por consiguiente, si una variable debe retener su valor entre sucesivas llamadas a un procedimiento, debe definirse fuera del procedimiento. Como consecuencia, las variables locales consumen recursos de memoria solamente durante la ejecución de la secuencia de instrucciones de su procedimiento. Tan pronto como el control pasa a la instrucción que sigue a la llamada al procedimiento se libera la memoria de las variables locales.

```
def calculos (a,b):
  global x3
  print(x3)   # x1 y x2 no se conocen en el procedimiento
  ya que no se han declarado globales
  x1 = 10      # no declarada global
  x2 = 5       # no declarada global
  x3 = 0     # modifica el valor de la variable declarada
  global
  a -=1          # resta una unidad al primer parámetro
  b +=2         # suma dos unidades al segundo parámetro
  print(a,b)       # escribe los nuevos valores de los
  parámetros
  print(x1, x2, x3)   # escribe el valor de las variables
  globales


  x1, x2, x3 = 3, 8, 5
  calculos(x1, x2)
  print(x1, x2, x3)
```

```
5
2 10
10 5 0
3 8 0
```

Figura 4.11. Ejemplo de ámbitos.

El procedimiento mostrado en la figura 4.11, ilustra estas ideas. Mediante asignación múltiple, se asignan valores a tres variables en la forma: x1, x2, x3 = 3, 8, 5. A continuación se efectúa una llamada al procedimiento en la forma: calculos(x1, x2). Tras la llamada al procedimiento se asigna al parámetro "a" el valor de x1, y al parámetro "b" el valor de x2. Dentro del procedimento, las variables x1 y x2 son desconocidas ya que se han declarado después del procedimiento y no se han declarado globales. Dado que la variable x3 se ha declarado global, x3 sí que es conocida dentro del procedimiento y su valor se puede modificar.

El cuerpo del procedimiento lo primero que hace es escribir el valor de la variable global x3, que es 5. Si intentásemos escribir los valores de x1 y x2 nos daría error ya que se han asignado después del procedimiento, y no se han declarado como globales en el procedimiento, por lo que no las conoce.

A continuación se asignan valores a las variables x1, x2 y x3. Pero las variables x1 y x2 no son las mismas que las que se han utilizado fuera del procedimiento, sino que son locales al procedimiento y aunque se les dé un valor dentro, no afecta para nada a las variables exteriores del mismo nombre ya que son de un tipo inmutable.

En las siguientes líneas se decrementa el valor de "a" en una unidad, y se incrementa el valor de "b" en dos unidades. Como "a" valía 3, pasa a valer 2 y como "b" valía 8, pasa a valer 10. Esto hace que se imprima 2 y 10. Finalmente se escribe el valor de las variables locales x1 (10), x2(5) y la variable global x3 (0). Con esto ha terminado el procedimiento y han desaparecido las variables locales x1 y x2, así como "a" y "b". Cuando se ejecuta la última línea: print(x1, x2, x3), se escriben los valores de las variables x1 y x2 exteriores al procedimiento (3 y 8) y el valor de la variable x3 que se ha declarado global y se ha modificado en el procedimiento (0).

4.7. Ventajas de la localidad

Es una buena práctica de programación el utilizar localmente las variables en los procedimientos y funciones. Esto confina su existencia al procedimiento en el que tienen significado. Para hacer uso de una variable global, hay que hacer una declaración explícita del hecho de que sea global.

El uso de variables locales presenta las siguientes ventajas:

- La declaración está textualmente cerrada para su uso en el ámbito del procedimiento o función, lo que ayuda a la legibilidad del programa.
- Se elimina el uso local, inadvertido, de una variable global.
- Se minimizan los requerimientos de memoria porque las variables locales se liberan cuando termina el procedimiento al que pertenecen.

4.8. Generación de números aleatorios

Un fenómeno aleatorio es aquel que produce resultados que no podemos conocer de antemano, pero que presentan determinadas propiedades. Por ejemplo, consideremos el experimento aleatorio consistente en el lanzamiento de una moneda. No sabemos si saldrá “cara” o “cruz”, pero el experimento tiene una propiedad: la probabilidad de que salga “cara” es igual a la probabilidad de que salga “cruz”.

Se dice que se va a generar un número aleatorio, cuando no se puede prever cuál puede ser. Cuando la probabilidad de que sea uno u otro es igual, se dice que la distribución (de los números) es uniforme.

Evidentemente los conceptos de aleatoriedad y algoritmo son irreconciliables, por lo que hablaremos de números pseudo-aleatorios. Lo mas que podríamos esperar de un tal algoritmo es que fuese capaz de producir una secuencia muy larga de números en la que no exista un modelo fácilmente discernible. Además, si la distribución es uniforme, no se debe repetir ningún número.

Las bases matemáticas son muy sencillas. Simplemente se necesita es una elección de dos parámetros enteros, el módulo “m” y el multiplicador “a” en la fórmula de recurrencia:

$$z = (a * z) \% m$$

La secuencia debe comenzar con un valor inicial de z llamado semilla (*seed*). La elección de “m” y de “a” es crítica. Con $m=2^{31}-1=2\ 147\ 483\ 647$ y $a=7^5=16\ 807$ ” aparecen todos los números entre 1 y (m-1) exactamente una vez, antes de que la secuencia se repita (distribución uniforme). Como en la práctica no se necesitan números tan grandes, se normaliza r=z/m. De esta forma los números aleatorios están en el intervalo 0<r<1. Según esto, el algoritmo de generación se escribiría:

$z = (a * z) \% m$

return z/m

Sin embargo, aparece un problema y es que es fácil ver que el producto a*z fácilmente puede exceder el rango de los enteros representables con 32 bits.

Schrage propuso una modificación del anterior algoritmo que supera esta dificultad. Para ello se seleccionan nuevas constantes q={mayor entero no mayor que m/a}=127 773 y r=m % a=2 836 y se computa de acuerdo con el siguiente algoritmo:

```
g= a*(z % q)-r*(z // q)
if  g>0:
        z= g
else:
        z=g+m     que z>0
return z/m
```

Vamos a escribir una función para la generación de un número aleatorio. La variable z sobre cuyo valor opera la función se supone declarada en un contexto más amplio y, por tanto, se ha declarado global para que sea utilizable.

```
Def uniforme():
  global z
  a, m = 16807, 2147483647
  q = m//a
  r = m%a
  gamma= a*(z % q)-r*(z // q)
  if gamma>0:
    z= gamma
  else:
    z=gamma+m
  return z/m
z = 1
for i in range(1,10):
  print(uniforme())
```

```
7.826369259425611e-06
0.13153778814316625
0.7556053221950332
0.4586501319234493
0.5327672374121692
0.21895918632809036
0.04704461621448613
0.678864716868319
0.6792964058366122
```

Figura 4.12. Código para generar una sucesión de nueve números pseudo-aleatorios.

4.9. Anidamiento de ámbitos

Un procedimiento puede contener declaraciones anidadas de procedimientos. En analogía con las variables locales, los procedimientos así definidos son objetos locales en el ámbito del procedimiento que los contiene. En el siguiente ejemplo se muestra un procedimiento anidado. En el cuerpo de ese procedimiento figuran:

```
a -=1

b +=2

print(a,b)

dec(a, b)
```

Vemos que hay una llamada al procedimiento dec(c,d). este está declarado antes de la llamada para que lo pueda reconocer. En su interior, figuran:

```
c -=1

d -=1

print(c,d)

inc(c, d)
```

En donde el procedimiento inc(c,d) está declarado previamente a su llamada. Analizar el resultado obtenido que se muestra a continuación del código.

```
def anidados(a, b):
  a -=1
  b +=2
  print(a,b)
  def dec(c, d):
    c -=1
    d -=1
    print(c,d)
    def inc(e, f):
      e +=2
      f +=2
      print(e, f)
    inc(c, d)
  dec(a, b)
x1, x2 = 2, 4
anidados(x1, x2)
print(x1, x2)
```

```
1 6
0 5
2 7
2 4
```

Figura 4.13. Procedimientos anidados.

4.10. Efectos colaterales

Veamos el efecto de trabajar con variables globales las funciones. Consideremos la función de cálculo de un número aleatorio:

```
def uniforme():
  a, m = 16807, 2147483647
  q = m // a
  r = m % a
  g=a*(z % q)-r*(z // q);
  if g>0:
    z=g
  else:
    z=g+m
  return z/m
```

Figura 4.14. Efectos colaterales.

El efecto colateral de una llamada a uniforme es el cambio de la variable global z. Consideremos las secuencias:

```
z=1
x=z + uniforme()
```

```
z=1
x=uniforme() + z
```

En el primer caso, el valor de x es 1.0000078263692593. En el segundo es 1.6807000E+04 lo que aparentemente desafía la propiedad conmutativa de la suma. Hay que hacer especial hincapié en que la utilización de variables globales pueden producir efectos colaterales por lo que es una práctica no aconsejable.

```
z = 1
x = z + uniforme()
  print(x)
z = 1
x = uniforme() + z
print(x)
```

```
1.0000078263692593
16807.00000782637
```

Figura 4.15. Efectos colaterales.

4.11. Recursividad

Se ha visto que desde una función se puede llamar a una segunda función. Dos casos particulares curiosos son:

- El caso en que desde la segunda función se llama a la primera.
- El caso en que desde una función se llama a sí misma.

En ambos casos se habla de recursividad. Por ejemplo, vamos a escribir la función factorial en forma recursiva y en forma iterativa. Para escribir de forma recursiva el cálculo del factorial debemos pensar en la definición de factorial de forma recursiva. La definición de factorial es:

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

Es decir, el factorial de 0 es 1 y, en general, el factorial de n es n multiplicado por el factorial de (n-1). Se trata de una definición recursiva ya que se define el factorial de n utilizando en su definición en factorial de (n-1).

```
def factorial (n):
    if n == 0:
        return (1)
    else:
        return (n*factorial(n-1))
```

```
def factorial (n):
    fact = 1
    while (n>0):
        fact = fact*n;
        n = n-1
    return (fact)
```

Figura 4.16. Algoritmo recursivo e iterativo para el cálculo del factorial.

Veamos cómo funciona. Supongamos que hay que calcular el factorial de 4.

- Como 4 es distinto de 0, la función factorial de 4 devuelve 4 por factorial de 3: con lo que aparece una llamada a la función factorial con valor 3 del parámetro.
- Al llamar a factorial(3), como 3 es distinto de cero, la función devuelve 3 por factorial de 2. De nuevo se produce una llamada a la función factorial con 2 como parámetro.
- Al llamar a factorial(2), como 2 es distinto de cero devuelve 2 por factorial de 1. De nuevo se produce una llamada a la función factorial con 1 como parámetro.
- Al llamar a factorial(1), como 1 es distinto de cero, devuelve 1 por factorial de cero. De nuevo se produce una llamada a la función factorial con 0 como parámetro.

- Al llamar a factorial(0) devuelve 1 y termina el programa. Con esto, el resultado es:

$$4 \cdot \text{factorial}(3) = 4 \cdot 3 \cdot \text{factorial}(2) = 4 \cdot 3 \cdot 2 \cdot \text{factorial}(1)$$
$$= 4 \cdot 3 \cdot 2 \cdot 1 \cdot \text{factorial}(0) = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 24$$

4.12. Captura y tratamiento de excepciones

En los programas pueden aparecer errores en tiempo de ejecución que dan lugar a la generación de las conocidas como “Excepciones”. Ejemplo de esos errores pueden ser divisiones por cero, operaciones con tipos que sean incompatibles, etc. Por ejemplo, si hacemos:

```
>>> a, b=5.0, 0

>>> a/b

Traceback (most recent call last):

  File “<input>”, line 1, in <module>

ZeroDivisionError: float división by zero
```

Se podría usar una estructura de control alternativa *if* para controlar ese tipo de problemas, pero resulta engorroso para el código resultante. Es mucho más interesante hacer uso de las denominadas “Excepciones”, que tienen la forma: *try … except*. La traducción podría ser:

intenta (ejecutar)

  <sentencias con posible error>

excepto (si aparece un error)

  <sentencias tratamiento error>

Este mecanismo permite al programa atrapar los errores, es decir, detectar que se ha producido un error y actuar en consecuencia al objeto de que no se detenga el programa La idea es que el intérprete de Python intente ejecutar <sentencias con posible error> excepto si aparece un error, en cuyo caso se sustituyen por <sentencias tratamiento error>.

Por ejemplo:

```
a, b = 7, 0
try:
  a/b
except:
  if b == 0:
    print(“La solución es infinito”)
```

Figura 4.17. Captura y tratamiento de excepciones.

Se pueden usar varias cláusulas *except*, una para cada error que se vaya a tratar. Así, en el ejemplo anterior, no se ha tratado un posible caso de 0 dividido entre 0, que sabemos que es indeterminado. Se podría hacer:

```
a, b = 0, 0
try:
    a/b
except:
    if b == 0 & a ¡= 0:
        print(“La solución es infinito”)
    if a == 0 & b == 0:
        print(“El resultado es indeterminado”)
```

Figura 4.18. Con dos cláusulas except.

Al objeto de atrapar el error, se inserta el código susceptible de producir un error entre las palabras clave *try* y *except*. La forma de proceder es la siguiente: El programa intenta ejecutar las sentencias incluídas entre *try* y *except*. Si se produce un error, el programa ejecuta las sentencias que siguen a *except*.

Otra sintaxis permite atrapar un tipo de excepción dado poniendo tras la palabra clave *except* el tipo de excepción que se desea atrapar. Por ejemplo:

```
except ZeroDivisionError
    print(“División por cero”)
```

Esta sintaxis sigue el esquema:

```
try:
    # sentencias a proteger
except <excepción tipo 1>:
    # a ejecutar en el caso de <excepción tipo 1>
except <excepción tipo 2>:
    # a ejecutar en el caso de <excepción tipo 2>
except <excepción tipo 3>:
    # a ejecutar en el caso de <excepción tipo 3>
except:
    # a ejecutar en el caso de error distinto a los anteriores
else:
    # ejecutar si no aparece ningún error.
```

Siendo optativas las dos últimas cláusulas. Si no se especifica la última cláusula *except* y la excepción que se lanza no corresponde a ninguna de la lista anterior, el programa se detendrá a menos que el error se atrape posteriormente.

A continuación se da una lista no exhaustiva de algunos tipos de errores en Python.

Excepción	Explicación
ArithmeticError	Se ha producido un error en una operación aritmética.
FloatingPointError	Se ha producido un error en una operación con números reales.
IndexError	Se ha utilizado un índice erróneo para acceder a un elemento de una lista.
KeyError	Se ha utilizado en un diccionario una clave inexistente.
NameError	Se ha utilizado una variable o función inexistente.
OverflowError	Un cálculo realizado con enteros o reales sobrepasa la capacidad del ordenador.
ZeroDivisionError	Error de división por cero.

Figura 4.19. Recopilación de algunas excepciones.

4.13. Lanzamiento de una excepción

Cuando una función detecta un error, puede lanzar una excepción por medio de la sentencia *raise*. Así, en el ejemplo que sigue, se compara x con 0 y lanza la excepción ValueError si el valor de x es 0. La excepción se atrapa posteriormente.

```
def inversa(x):
  if x == 0:
     raise ValueError
  y = 1.0 / x
  return y
try:
  print(inversa(0))# error
except ValueError:
  print("Error")
```

A veces puede resultar útil asociar un mensaje a una excepción para que pueda dar pistas del fallo

```
  def inversa (x):
      if x==0:
          raise ValueError("Valor distinto de cero")
      y=1.0/x
      return y
  try:
      print(inversa(0)) #error
  except ValueError as exc:
      print("Error, mensaje: ",exc)
```

En unidades posteriores se ampliará el tratamiento de las excepciones tras haber estudiado la herencia en programación orientada a objetos.

4.14. Diseño descendente

Vamos a tratar de una forma sencilla el diseño descendente. Los problemas complejos suelen conllevar una gran cantidad de detalles que pueden ocultar su solución a alto nivel. Una vez más decimos que los árboles ocultan el bosque. Por ejemplo, la clasificación de los animales en vertebrados e invertebrados representa una abstracción, en virtud de la cual se destaca alguna característica relevante y se ignoran todas las demás. La idea será:

- Destacar los detalles relevantes.
- Ignorar los irrelevantes.

Esto permite estudiar fenómenos complejos siguiendo un método jerárquico, es decir, por sucesivos niveles de detalle. Por eso, cuando se debe realizar una tarea compleja de programación, en la que es fácil perderse en los detalles de las estructuras de datos o en el desarrollo de los algoritmos, es conveniente proceder con niveles de resolución cuidadosamente escogidos; primero buscando una solución amplia de la tarea en términos de sub-tareas, a continuación especificando las acciones de la sub-tarea, posiblemente posponiendo detalles para un posterior refinamiento.

Esta forma de hacer las cosas se conoce como “diseño descendente”, ya que va desde los niveles de menos detalle o más generales, a los más específicos. En ocasiones esta técnica se combina con un diseño ascendente que busca solucionar pequeños problemas hasta llegar a la solución del problema en cuestión.

En este proceso de refinamientos sucesivos, el procedimiento y la función es una herramienta indispensable. Si las tareas se realizan mediante procedimientos con identificadores (nombres de las funciones) adecuados a la función que realizan, la estructura interna del programa aparece muy visible y las tareas las pueden realizar diferentes grupos de programadores.

El módulo añade un nivel de estructura posterior a través de las relaciones de importación.

RESUMEN

- Un procedimiento es una instrucción que se crea para sustituir una secuencia de instrucciones que tenga sentido en sí misma, a la que se da cierto nombre.
- Los datos se pasan al procedimiento por medio de parámetros.
- Los procedimientos pueden devolver resultados por medio de parámetros si dichos parámetros son de tipos mutables.
- Las funciones devuelven los datos por medio de una cláusula *return*.
- En realidad, en Python tanto a procedimientos como a funciones se les suele llamar funciones. Por eso, se habla de funciones propias y funciones procedimiento.
- Una llamada a procedimiento consiste en escribir el nombre del procedimiento como si fuese una instrucción, seguido de la lista de parámetros entre paréntesis.
- Las funciones se utilizan mediante una operación de asignación por medio de la cual devuelven los resultados a la variable o variables a las que se asignan.
- En una función, la asignación múltiple proporciona el mecanismo que permite que haya varios parámetros de salida.
- Un procedimiento tiene: cabecera y cuerpo.
- La cabecera comienza por la palabra def seguida del nombre (identificador) del procedimiento. Tras él, entre paréntesis la lista de parámetros del procedimiento. Termina con el carácter dos puntos.
- El cuerpo contiene las instrucciones que resuelven el problema para el que se ha diseñado el procedimiento.
- Para poder usar una variable global a un procedimiento, hay que declararla dentro del procedimiento como: global.
- Recursividad es que una función se pueda llamar a sí misma.
- Las excepciones permiten al programa atrapar los errores, es decir, detectar que se ha producido un error y actuar en consecuencia al objeto de que no se detenga el programa.
- Las excepciones tienen la forma: *try … except* insertando el código susceptible de producir un error entre ambas. El programa intenta ejecutar las sentencias incluidas entre try y except. Si se produce un error, el programa ejecuta las sentencias que siguen a except.
- Cuando una función detecta un error, puede lanzar una excepción por medio de la sentencia *raise*.