Artículo invitado

Paseo por la programación estructurada y modular con Python

Rosalía Peña
Universidad de Alcalá

Resumen

El lenguaje Python presenta particularidades que pueden dificultar un primer acercamiento y ser fuente de quebraderos de cabeza, incluso a programadores expertos en otros lenguajes, si no son entendidas. Este documento pretende presentar de una manera práctica la parte procedimental del lenguaje a programadores de otros lenguajes, indicando similitudes y explicando diferencias, con el objetivo de reducir los tiempos de aprendizaje.

Palabras clave: Programación, Python.

1. Introducción

En el contexto del paradigma procedimental y adoptando una metodología acorde con la programación estructurada y modular encontramos muchas similitudes en los constructores de distintos lenguajes (Basic, COBOL, FORTRAN, C, Pascal, PL/1, etc.): llámense *record* o *struct*, escríbase { } o *begin end*, := o =, barajan conceptos equivalentes. Los servicios concretos varían (por ejemplo: tipo *complex*, enumerados) si no están disponibles, habrá que programarlos. Ciertamente hay otras diferencias: declaraciones de tipo implícito en FORTRAN (metodológicamente evitables) o explícitas en el resto. El constructor *cadena* es una especialización de arreglo en Pascal y no en C. En unos (FORTRAN, Pascal) existe una palabra reservada para distinguir subprogramas que hacen cosas (procedimientos), de los que calculan (funciones). En los otros, las funciones que devuelven nulo hacen el papel de los procedimientos. FORTRAN especifica si el parámetro de un subprograma es de entrada/salida/entrada+salida, en Pascal no se distingue entre los dos últimos, C solo tiene parámetros de entrada, pero se consigue el comportamiento del paso por referencia a través de punteros. La precedencia de operadores es distinta, requiriendo o no paréntesis. FORTRAN no tiene el bucle *uno-a-n* (*repeat-until* en Pascal o *do-while* en C). El *case* de C requiere un *break* tras cada caso, para que se comporte como alternativa múltiple (tal como la entiende la programación estructurada), en otros lenguajes está disponible directamente, y un largo etc. de diferencias conocidas; pero, cualquier programador de uno de estos lenguajes, dotado de una "chuleta" con la sintaxis de otro (nuevo para él), puede traducir sus programas, prácticamente línea por línea.

Esta equivalencia entre los lenguajes no es aplicable a Python. No encontramos en su base nada parecido a los constructores de tipo *array* y *struct* (aunque dispone de bibliotecas que los simulan). Peor aún, la gestión de memoria es completamente diferente a la de otros lenguajes, de modo que incluso el concepto de asignación difiere. Estas singularidades pueden dificultar el aprendizaje de Python y generar pesadillas. El objetivo de este documento es facilitar el acercamiento de un programador de otros lenguajes a Python, aprovechando lo conocido por ser común, explicando las diferencias y previniendo errores de tiempo de ejecución.

2. ¿Quién es Python?

Python fue creado a principios de los noventa por Guido van Rossum en los Países Bajos. Es relativamente joven (Fortran 1957, Pascal 1970, C 1972, Modula-2 1978, Java 1991). Toma características de lenguajes predecesores, incluso, compatibilizando la solución de varios de ellos. Por ejemplo, habilita tres formas de imprimir el valor de una variable: desde el entorno interactivo escribiendo su nombre (como en Basic), usando la función *print*, con concatenación de elementos (al estilo del *write* de Pascal) o bien con patrones de formato (al estilo del *printf* de C).

Lo administra la Python Software Foundation y se distribuye bajo licencia de software libre. Es multiplataforma. Los archivos de programa tienen extensión ".py".

Es un lenguaje basado en pocos y potentes constructores ortogonales. Está libre de elementos mantenidos por compatibilización con versiones anteriores (hay dos versiones "estables": la 2.7 y la 3.4.2. No se ha mantenido la compatibilidad hacia atrás). Promueve la creación de código legible. Es interpretado. Tiene tipos dinámicos, conversión explícita entre todos los tipos de datos, gestión dinámica de la memoria (recolector de basura). Maneja excepciones.

Es multiparadigma: soporta programación imperativa, orientación a objetos, y, en menor medida, programación funcional.

Es ampliamente empleado en educación [1] y de presencia creciente en el entorno laboral [5], gracias a la disponibilidad de aceleradores y sus bibliotecas versátiles.

Indiscutiblemente, Python no pasa desapercibido. Sus admiradores lo son fervientemente. La documentación y código generados rezuman entusiasmo y un inusual grado de informalidad, rindiendo culto al humor. No en vano, el nombre del lenguaje es un homenaje al famoso grupo cómico británico.

3. Declaraciones, organización y documentación

No existe cabecera de programa o módulo, por lo que el consabido primer proyecto queda como:

```
>>>print('hola mundo')
'hola mundo'
```

Incluso menos (desde el intérprete no necesita *print*), es suficiente:

```
>>> 'hola mundo'
'hola mundo'
```

¡Hecho!

No existe declaración explícita de tipo (como sucede en Perl y parcialmente en FORTRAN, pero a diferencia de C y Pascal). Los tipos son dinámicos: en distintos momentos del programa, una variable puede ser de distintos tipos. Me apoyo en la función interna *type*, para ejemplificarlo. La variable `b` empieza siendo de tipo *int*, y la nueva asignación provoca su modificación a *str*.

```
>>> b=5
>>> type(b)
<class 'int'>
>>> b='cadena'
>>> type(b)
<class 'str'>
```

Al igual que en C, y a diferencia de Pascal, los nombres de las variables son sensibles al caso de la letra (mayúscula/minúscula), de modo que `B` y `b` son variables distintas.

Se comenta (se hace transparente al intérprete) el contenido hasta fin de línea (con `#`) y varias líneas, encerrándolas entre dos grupos de 3 comillas dobles `""" """`.

No hay delimitadores de bloque (*begin end* o { }). Los bloques de ejecución se estructuran por sangrado, con normas estrictas (4 espacios o tabulación, pero no ambas, el *else*, justo debajo de su *if*, etc.). Esto es novedoso (no conozco otro lenguaje que lo haga). Al igual que Eiffel, no requiere terminadores de instrucción (los ";" de Pascal).

El patrón de la definición de función puede incluirse en cualquier parte del programa, pero antes de ser llamado. No existe especificación del tipo de función ni del de sus argumentos. Subsana esta falta de información fomentando el uso de buena documentación, al ofrecer un servicio de valor añadido: al empezar a escribir la llamada a la función, el editor muestra la documentación generada (entre triples comillas inmediatamente posterior a la instrucción *def*), como ejemplifica la Figura 1.

4. Subprogramas y bibliotecas

El lenguaje Python no distingue entre funciones y procedimientos, pero un subprograma puede devolver *None* (procedimiento), o una variable de cualquier tipo, incluso compuesto. Como los paréntesis son opcionales en las tuplas (de las que hablaremos más adelante), puede parecer que una función está devolviendo varios parámetros. Por ejemplo, en la biblioteca estándar se dispone de la función *divmod*, que devuelve el par (cociente, resto) de la división entera.

```
>>> div,resto=divmod(5,3)
>>> print('división=',div,' resto:=',resto)
división= 1  resto:= 2
```

Python proporciona funciones internas (*built-in*) [6], que se invocan en la forma habitual de la programación procedimental (variable=función(arg)), y funciones específicas de cada uno de los tipos de datos, que se invocan como método del objeto solicitante (objeto.metodo(arg)), al estilo de la OO.

```
>>> abs(-3)   #interna
3
>>> c='hOlA'
>>> c.capitalize()  #de clase
'Hola'
```

La frase «Python viene con las pilas puestas» [4] hace referencia a la completitud de su biblioteca estándar, que incluye: interfaz a S.O., email, XML, gráficos, paquetes matemáticos, fechas_tiempo, compresión de datos, control de calidad, etc. Para usar una función de biblioteca hay que importarla:

```
>>> import datetime
>>> print(datetime.date_today()) # de
    biblioteca
```

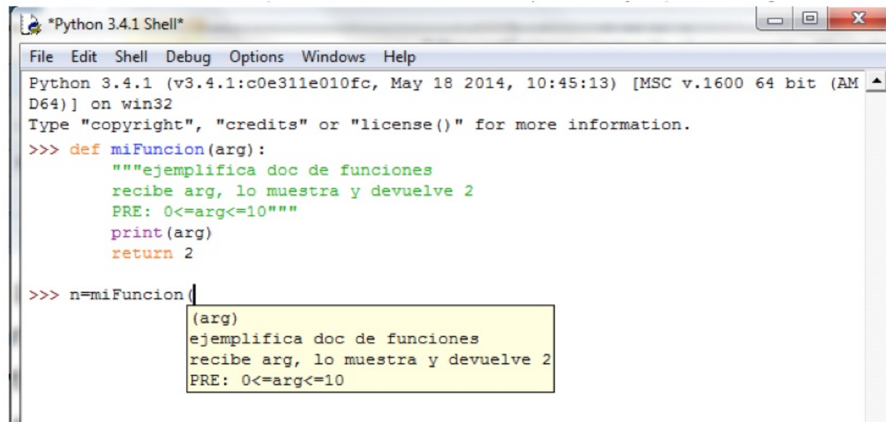


Figura 1: El editor muestra la documentación de la función.

```
2014-10-24
```

Hay muchas bibliotecas generadas por la comunidad de software libre que cubren diversos aspectos.

5. El concepto de variable

Todo valor en Python es un objeto (el número 5, por ejemplo). A diferencia de lo que ocurre en otros lenguajes (FORTRAN, Pascal, C...), una variable en Python no es un contenedor de un valor, sino un puntero al valor. La asignación no produce una copia sino una referencia al objeto al que se asigna. Nos apoyamos para confirmarlo en la función *id*, que devuelve el identificador del objeto (digamos su posición de memoria).[1]

```
>>> a=5
>>> b=a
>>> id(5)
1537974848
>>> id(a)
1537974848
>>> id(b)
1537974848
```

los tres (a, b y 5) son el mismo objeto. Es más:

```
>>> c=5
>>> id(c)
1537974848
```

Python busca el objeto "5" y lo reutiliza[2]. Al reasignar la variable, por supuesto, pasa a apuntar al nuevo valor. Siguiendo con el anterior estado de memoria:

```
>>> b=6
>>> id(b)
57012080
```

La Figura 2 muestra la situación de la memoria antes y después de esta asignación.

6. Conjunto de instrucciones ejecutables

Estructuramos el estudio de las instrucciones de acuerdo al número de veces que se ejecuta el bloque (exactamente una vez, cero o una y n veces). Al final del documento, en la Figura 5, se presenta un esquema global del lenguaje.

Instrucciones que se ejecutan exactamente una vez:

- La asignación es, por excelencia, la representante de este grupo. El diferente concepto de variable no modifica el comportamiento esperado de la asignación en los tipos simples y en los estáticos; pero sorprende al programador procedente de otros lenguajes en los tipos "mutables" (que veremos más adelante).
- Las funciones pueden ser evocadas como los procedimientos en Pascal, es decir, su llamada es una instrucción (y corresponden por tanto al grupo "1 vez").
- La gestión de excepciones: *try* ... *except* ... *finally* (que se verá más adelante), cierran este apartado.

Instrucciones que ejecutan su bloque 0-1 vez (alternativas):

- Python dispone de *if* simple y doble (con *else* opcional)

```
def diaSemana1 (num):
    if num==1: nom='lunes'
```

[1] Algunos de los ejemplos de este documento fueron sugeridos por [2]

[2] Cierta inseguridad proporciona detectar que el localizador de objetos se pierde con valores mayores y no reutiliza el objeto: Tras a=500 y *b=500*, id(a) $\neq$ id(b).

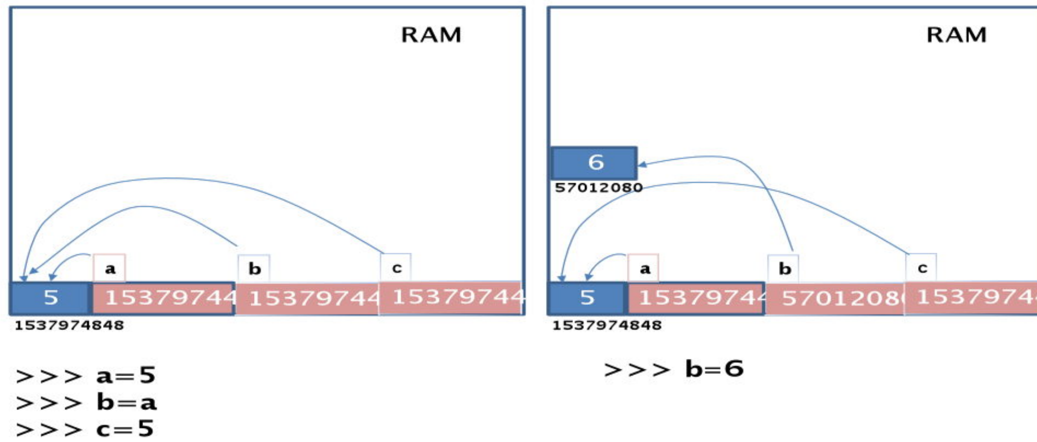


Figura 2: Estado de la memoria tras asignaciones.

```
    else:
        if num==2: nom='martes'
        else:
            if
            ...
              else: nom='domingo'
    return nom
```

- *elif* evita códigos excesivamente sangrados ante una serie de if anidados. Apréciese el sangrado exigido[3] que indica el bloque

```
def diaSemana2 (num)
    if num==1: nom='lunes'
    elif num==2: nom='martes'
    elif
    ...
    else:nom='domingo'
    return nom
```

- Python no dispone de instrucción alternativa múltiple (*switch* o *case*). A veces se consigue la funcionalidad del *case* seleccionando un elemento de una secuencia. El siguiente ejemplo usa una tupla (un arreglo usaríamos en otros lenguajes):

```
nombreDia=('lunes','martes',...,'domingo
    ')
def diaSemana3 (num):
  return nombreDia(num-1):
```

Esta es una opción más elegante y mantenible que el propio *case*. A modo de comparación se incluye el código equivalente en C usando el *switch*.

```
switch (num) {
  case 1: ud='lúnes' break;
  case 2: ud='martes' break
  ...
  case 7: ud='domingo' break;
}
```

Instrucciones que ejecutan su bloque n veces (bucles):

- Python dispone de bucle *while*, con el comportamiento esperado (0-n controlado por condición), y
- un *for* muy potente en el recorrido de las estructuras de datos compuestas del lenguaje, pero algo distinto de lo esperado. Lo estudiamos con dichas estructuras.
- No dispone de bucle (1-n), al igual que FORTRAN.[4]

7. Tipos de datos (o el misterio de la inmutabilidad)

Estructuramos este apartado en tipos simples y compuestos. Dentro de los compuestos, atenderemos primero a los estáticos y posteriormente a los de tamaño variable.

Tipos simples: Python dispone de enteros, reales y booleanos. No existe el tipo carácter. Comprobamos en el intérprete que un solo carácter ya es una cadena:

```
>>> c='a'
>>> type(c)
<class str>
```

[3] Apreciemos la elegancia de Python en los dos ejemplos siguientes, que son los más largos del documento. Sin artefactos, orientados únicamente al traductor, habituales en otros lenguajes, un programa Python es prácticamente seudocódigo, incrementando la legibilidad y mantenibilidad.

[4] Oportunidad desaprovechada.

La documentación recomienda el uso de paréntesis en las expresiones, evitando extenderse en la descripción de la precedencia de operadores[5]. Proporciona la conversión automática de tipos numéricos hacia el de mayor precisión, en la línea de lo que hacen otros lenguajes:

```
>>> 1+3  #entero+entero -> entero
4
>>> 1.0+3  #real+entero -> real
4.0
>>> 5//3  #división entera
1
>>> 5/3  #división real -> real
1.6666666666666667
```

Tipos compuestos: No están disponibles los constructores de tipo habituales de otros lenguajes: arreglo y registro (sí lo están a través de bibliotecas específicas). Aparte de un tipo *complex*, similar al complejo matemático de FORTRAN, que no requiere especial comentario, Python dispone de tres tipos que podríamos llamar estáticos o "de tamaño fijo" (denominados en la bibliografía como *inmutables*), que son:

- **Conjunto congelado** *frozenset*: colección de elementos, potencialmente de distinto tipo (mientras estén entre los simples e inmutables), no repetidos y sin orden entre sí:

  ```
  >>> congelado=frozenset({3,5,1,1})
  >>> congelado
  frozenset({1, 3, 5}) #elimina repetido
  >>> type(congelado)
  <class 'frozenset'>
  ```

  (¡El adjetivo en este tipo, ya nos avisa de se avecina una patata caliente!).

- **Cadena** *str*: secuencia de caracteres:

  ```
  >>> cadena='buenos días'
  >>> type(cadena)
  <class 'str'>
  ```

- **Tupla** *tuple*: secuencia de 0, 1 o n elementos, potencialmente de distinto tipo[6]:

  ```
  >>> t1=(1,'a',3,3)
  >>> type(t1)
  <class 'tuple'>
  >>> t1
  (1, 'a', 3,3)
  ```

Los elementos de las secuencias (*str* y *tuple*) se acceden individualmente por su índice. El índice empieza en cero[7]:

```
>>> cadena
'buenos días'
>>> cadena[1]
'u'
```

Los delimitadores de cadena pueden ser comillas simples o dobles (en parejas no intercambiables). La cadena vacía, como se puede esperar, es ' '.

La tupla vacía se consigue con paréntesis vacíos (). En tuplas de mayor longitud, los paréntesis son opcionales. En `tUnitaria` se han omitido. Los elementos de la tupla se separan por comas. Es necesario que exista al menos una coma en la tupla, como en `tUnitaria`. Aprecie en el ejemplo siguiente que `noTupla` es de tipo un entero (los paréntesis forman parte de las expresiones, ergo no pueden caracterizar al tipo tupla). `tVariopinta` contiene datos personales de un deportista. El último elemento es, a su vez, una lista de los deportes que practica Ana.

```
>>> tVacia=()
>>> tUnitaria='hola',  # OJO la ","
>>> type(tUnitaria)
<class 'tuple'>
>>> noTupla=(1)  #sin "," no es tupla
>>> type(noTupla)
<class 'int'>
>>> tVariopinta=(22918674,'Ana','Perez'
    ,28,1.75,('ciclismo','natación'))
>>> type(tVariopinta[6])
<class 'tuple'>
```

Mientras que los tipos secuencia (cadena y tupla) conservan el orden en que se introdujeron los elementos, en el conjunto congelado no ha de mantenerse el orden (como corresponde al concepto de conjunto: sin orden).

En el ejemplo se introdujeron dos "1" al definir `congelado`. El elemento repetido se eliminó (como corresponde a un conjunto: sin repetidos). Sin embargo en la tupla `t1` se han mantenido dos "3".

Python dispone de funciones internas para convertir de uno a otro tipo. El ejemplo siguiente usa estas funciones para aislar los caracteres de una cadena, devolviendo una tupla:

```
>>> cadena
'Hola'
>>> tAisla=tuple(cadena)
>>> tAisla('H', 'o', 'l', 'a')
```

y en el siguiente se usan para eliminar los elementos repetidos de una tupla:

```
>>> t1=3,1,2,3
>>> t1
(3,1, 2, 3)
>>> t1=tuple(frozenset(t1))  #composición
>>> t1
```

[5] Ganando legibilidad y evitando confusiones.
[6] Incluidos tipos mutables como elemento.
[7] Desafortunadamente, pues el mundo real suele empezar en uno.

```
(1, 2, 3)
```

La instrucción *for* recorre los elementos de una tupla, conjunto o cadena de forma elegante:

```
>>> semana = ('L','M','X','J','V','S','D')
>>> for dia in semana:
  print (dia)
L
M
X
J
V
S
D
```

(de forma similar a algunos usos en Perl de este tipo de bucle).

La función *len* devuelve el número de elementos que tiene una cadena, tupla o conjunto (recuerda que uno de los elementos de `tVariopinta` es a su vez compuesto):

```
>>> len(congelado)
3
>>> len(tVariopinta)
6
```

Existen operadores y funciones que proporcionan servicios específicos a la clase (tipo), por ejemplo, contar el número de veces que aparece un elemento en una secuencia; las operaciones típicas de conjuntos (pertenencia, unión, intersección, inclusión. . .), etc.:

```
>>> conj1=frozenset({'inf',1,0,5,('a',1)})
>>> len (conj1)
5
>>> 'inf' in conj1 #pertenencia
True
>>> conj2=frozenset{1,2}
>>> conj1 & conj2 # como operador
{1}
>>> conj1.union(conj2) #como método
{0, 1, 2, 5, ('a', 1), 'inf'}
>>> conj3=frozenset{2}
>>> conj3<=conj2 #inclusión,disjuntos
True
>>> conj3==conj2
False
>>> conj2.issubset(conj3)
False
```

Dispone de las muy conocidas funciones de cadenas (*upper*, *lower*, *concat* o +) y otras no tan habituales.

Las tuplas son estructuras muy potentes. Te invito a averiguar la salida del siguiente programa:

```
x=3
y=6
x,y=y,x
print (x)
print(y)
```

¡¡¡EXACTO!!!! "6, 3". ¡Se acabaron contenedores auxiliares!

PEEERO. . . (ya nos temíamos que existía un pero, con lo de "congelado"..), el valor de una variable de cualquiera de estos tres tipos sólo puede ser cambiado por reasignación, no pueden adicionársele elementos, ni modificar los existentes:

```
>>> cadena='saludo'
>>> cadena[0]='S'
Traceback (most recent call last):
File "<pyshell#73>", line 1, in <module>
     cadena[0]='S'
TypeError: 'str' object does not support
    item assignment
>>> cadena='Saludo' #si, puedo reasignar
```

Al reasignar, la variable referencia a un objeto diferente. El objeto antiguo ('saludo') se destruye (si nadie más lo apuntaba) y el nuevo ('Saludo') se crea. Y donde dijimos tipos de "tamaño fijo" pudimos decir de "tamaño y valor fijo". Por ello llama Python a estos tipos "inmutables". Decir que un tipo es estático o que su tamaño es fijo se digiere bien, pero decir que una variable es inmutable suena anacrónico (por eso dimos un par de rodeos).

Si en un entorno se califica a un grupo (de tipos de datos, en nuestro caso) con un adjetivo (inmutable), habrá otro grupo que no responde a ese calificativo. En efecto, Python dispone de tres tipos dinámicos (mutables), que pueden cambiar el valor (y el tamaño) de sus elementos individualmente, insertar y eliminar elementos, sin reasignación del objeto como un todo. Dos de estos tipos son "los hermanos mayores" de conjunto congelado y de tupla: conjunto (*set*) y lista (*list*) respectivamente.[8] El tercer tipo, diccionario (*dict*) no tiene su contrapartida "congelada" (ni se prevé implementarla [4]).

Set: colección de elementos (sin orden y no repetidos), potencialmente de distinto tipo simple o inmutable, (hasta aquí, nada nuevo respecto a *frozenset*),

```
>>> conj1={'infinito',1,0,5,('a',1)} #
    sin frozen
```

al que se le pueden añadir nuevos elementos o eliminar existentes:

```
>>> c3= {2}
>>> c3.add(5)
>>> c3.remove(2) #si 2 no existe ->
    error
>>> c3
{5}
```

Los diccionarios también se delimitan con llaves. Una pareja de llaves vacía es un diccionario. El conjunto vacío se crea con la función *set*.

[8] Más sistemático hubiera sido que las tuplas se llamaran "listas congeladas".

```
>>> otraCosa={}      #OJO es diccionario
>>> type(otraCosa)
<class 'dict'>
>>> conjVacio=set()
>>> len(conjVacio)
0
>>> type(conjVacio)
<class 'set'>
```

Lista: secuencia de elementos, potencialmente de distinto tipo (incluidos mutables), a la que se le puede eliminar elementos, añadir nuevos, y modificar valores individuales:

```
>>> lista=['hola',4, (1,2),{3,4}] #
    mutable {3,4}!
>>> type(lista)
<class 'list'>
>>> len(lista)
4   #2 elementos son dobles
>>> lista[0]='adios'  #modifico 1er
    elemento
>>> lista.append(2) #añado al final
>>> lista.insert(3,'gato')#añado en 3
>>> lista
['adios', 4, (1, 2), 'gato', {3, 4}, 2]
>>> lista.remove({3,4}) #quito 1era
    aparición
>>> lista.reverse()    #invierto lista
>>> lista
[2, 'gato', (1, 2), 4, 'adios']
```

Diccionario: colección (conjunto) de pares de clave-valor. La clave es de cualquier tipo inmutable[9]. Los valores pueden ser de cualquier tipo. En la Figura 3 (a) hemos documentado un sencillo gazpacho.

Se accede al valor de los elementos por su clave (como puede verse en la Figura 3 (b)). Puede variar el valor de los elementos individualmente, pero no la clave. Pueden eliminarse elementos y crear nuevos (como vemos en la Figura 3 (c)).

Cada tipo mutable dispone de una amplia colección de métodos, como ordenar, invertir lista, etc.

Cualquiera que haya programado estructuras dinámicas intuye que los tipos mutables de Python son muy consumidores de recursos. Eso justifica la presencia de la contrapartida estática, pagando algo de duplicidad, a cambio de eficiencia.

Consecuencias del concepto de asignación sobre los tipos mutables: Recordamos que en Python todo valor es un objeto y la asignación referencia al objeto, no hace una copia. Acabamos de ver que el contenido de variables de los tipos mutables se puede cambiar sin reasignar el objeto completo. Partimos de dos variables apuntando al mismo objeto y lo modificamos desde una de ellas:

```
>>> lista1=[1,2,3]
>>> lista2=lista1
>>> lista2[2]=5
>>> lista2              #sin sorpresa
[1, 2, 5]
>>> id(lista1)==id(lista2)   #mismo objeto
True              #¿lista1 vale?
```

Si cambia el valor de un objeto, por supuesto cambia el valor de cualquiera de las variables que referencian al objeto. Comprobémoslo (con el mismo estado de memoria):

```
>>>lista1
[1, 2, 5]
```

¡Pero... si no la he tocado! No comprender el mecanismo de la asignación, puede volver loco al programador. La reasignación del objeto mutable (como un todo), por supuesto, cambia el objeto al que referencia. Continuando con el ejemplo (en el mismo estado de la memoria):

```
>>> lista2=[2,3,4]
>>> id(lista1)==id(lista2)
False
```

El método *copy*, en cada uno de los tipos mutables, produce una copia del objeto que es lo equivalente a la asignación en otros lenguajes. Al ser variables independientes, no se producen efectos laterales.

```
>>> lista3=lista1.copy()
>>> id(lista3)==id(lista1)
False
>>> lista3
[2, 3, 4]
>>> conj1={'c','m','x'}
>>> conj2=conj1.copy()
>>> dict1={1:'lu',2:'ma',3:'mi'}
>>> dict2=dict1.copy()
```

Este método es innecesario en los tipos inmutables y no existe.

8. Paso de parámetros

Python no tiene el concepto de valor/referencia, pero sabemos que la variable es la referencia al objeto, no una copia del valor. Hemos visto que sólo se puede cambiar el valor de variables de los tipos simples y de los inmutables reasignándolas. La asignación es equivalente a crear una nueva variable (el colector de basura destruirá la vieja), por tanto, el comportamiento de estos tipos, cuando son pasados como argumento es como si fuera por valor.

Por el contrario, las variables de los tipos mutables (que, igualmente son una referencia al objeto) pueden cambiar de

[9] Incluidas tuplas, siempre que todos sus elementos sean inmutables. Sorprende que las claves pueden ser de distinto tipo. Supongo que le encontraremos utilidad: `>>> dicRarito = {1:1,'A':2}`

```
 >>> ingredientes={'tomate':(1,'Kg'), 'pepino':2,'sal':'1 cucharilla', 'aceite':.1,
    'oregano':'1 pizca'}
>>> type(ingredientes)
<class 'dict'>
```

(a)

```
>>> ingredientes['pepino']
2
>>> uTomates=ingredientes['tomate'][1]
>>> uTomates
'Kg'
```

(b)

```
>>> ingredientes["ajo"]='1 diente'                # añado ingrediente
>>> ingredientes['aceite']=ingredientes['aceite']*2    # duplico generosamente
>>> del ingredientes['sal']       # pero tengo hipertensión
>>> ingredientes{'pepino': 2, 'oregano': '1 pizca', 'ajo': '1 diente', 'tomate': (1,
     'Kg'), 'aceite': 0.2}
>>> del ingredientes          # ya no quiero gazpacho
>>> ingredientes
Traceback (most recent call last):
   File "<pyshell#11>", line 1, in <module>
         ingredientes
   NameError: name 'ingredientes' is not defined
```

(c)

Figura 3: Diccionarios: (a) declaración; (b) acceso; (c) gestión.

valor sin asignación (global del objeto), añadiendo o eliminando elementos o cambiando el valor de elementos individuales. Estos procesos cambian el contenido del objeto al que referencian, pero no la referencia del propio objeto; por lo tanto, los argumentos de tipos mutables, se comportan como los pasados por referencia (con *var* en Pascal) ante modificaciones parciales. Pura consecuencia del concepto de asignación.

En el siguiente ejemplo, comprobamos que el subprograma `subpro` tiene 3 parámetros. Reasigna valor a `nF` y `listF`, cambia el objeto y por tanto, las variables del programa principal `nR` y `listR` no son afectadas. Sin embargo, el parámetro real `setR` se ve afectado por la modificación de `setF` en el subprograma.

```
>>> def subPro (nF, lisF, setF): #
    parametros formales
        nF=4
        lisF=[1]  #asigna
        setF.add('patata')  #modifica
        return

>>> nR=0  #parametros reales
>>> lisR=[0]
>>> setR={0}
>>> subPro(nR,lisR,setR)
```

```
>>> nR
0
>>> lisR
[0]
```

9. Ámbito y visibilidad

Incluso, si el subprograma asigna un parámetro mutable a una nueva variable local, y modifica la local, el valor del argumento real quedará cambiado. El uso de la función *copy*, en vez de referencia directa evita este efecto.

Respecto al ámbito, las variables locales mantienen el comportamiento habitual en otros lenguajes: se destruyen al terminar la ejecución de la función, por tanto, no son accesibles desde el subprograma llamante.

En la dirección contraria, es posible acceder (consulta) a las variables globales desde un subprograma si no se modifica su valor pero, en cuanto se modifica, la variable pasa a ser local (se crea una nueva variable, y no permite el acceso previo), excepto que se indique expresamente que es modificación de la global (por ej: `global b=3`). Por tanto, Python protege de los efectos laterales del uso global con reasignaciones. Pero, como las mutables pueden cambiar su valor de

formas distintas que la reasignación "total de la variable", aparecerán efectos indeseables tras un inadecuado uso global. La ejecución del código:

```
def cajaNoTanNegra():
    lista[0]=3
    lista.append('a')
    print('dentro:', lista)

lista=[1,2,3]
print('antes',lista)
cajaNoTanNegra()
print('después',lista)
```

proporciona la salida:

```
antes [1,2,3]
dentro [1,2,3,'a']
después [1,2,3,'a']
```

10. Consideraciones de diseño en la resolución de problemas

Siempre que la estructura no requiera crecer usaremos la versión congelada.

En cualquier lenguaje que los tenga, los conjuntos (*set* o *frozenset*) son cómodos para evaluar la pertenencia, para iterar con sus elementos, pero en Python, ambas cosas se pueden hacer también fácilmente con tuplas (o listas), incluso concatenarlas:

```
>>> tupla =(1,2,3,4)
>>> 3 in tupla
True
>>> t2=4,5
>>> t3=tupla+t2
>>> t3
(1, 2, 3, 4, 4, 5)
```

Los conjuntos congelados (*frozenset*) son cómodos para eliminar los elementos duplicados de una secuencia, por ejemplo después de haber unido dos colectivos, o si resulta relevante al problema alguna de las operaciones clásicas de conjuntos.

Tuplas y listas son adecuadas para trabajar con colecciones de datos que mantienen un orden, cuando puede haber elementos repetidos y cuando la gestión conlleva un acceso secuencial. Los diccionarios por el contrario, soportarán mejor los datos cuyo tratamiento típico sea el acceso directo.

Hasta introducir la OO (que agrupa de forma natural datos y métodos de una clase de objetos), un gimnasio podría modelarse mediante un diccionario de deportistas, con clave su DNI (por ejemplo) y un deportista podría modelarse como una tupla de la secuencia de sus atributos; caber cabe, pero perdemos semántica del nombre del atributo en el código ¿Qué tal una lista de diccionarios, que a su vez pueden contener listas? Vayamos dándole vueltas.

11. Manejo de excepciones

En lenguajes como Pascal o Fortran con frecuencia el volumen del código escrito para manejar los potenciales errores es mayor que el del código "efectivo" de la tarea que se deseaba realizar. Python dispone de un gestor de excepciones fácil de entender y usar. La instrucción *try* permite capturar la aparición de una situación excepcional en tiempo de ejecución, evitando que aborte el programa y devolviendo al programador el control, que puede analizar el tipo de error acaecido (mediante la instrucción *except*) y obrar en consecuencia. En la Figura 4 se muestra un programa ejemplo y el resultado de su ejecución.

12. Transición entre paradigmas

Tras el precedente conocimiento será sencillo afrontar la OO en Python. La clase es el colector propuesto para abstraer todos los atributos (Python llama atributos tanto a las propiedades que tienen un valor, como a las que definen el comportamiento: métodos) de un tipo (clase) de objetos.

Uno de los problemas típicos en el cambio de paradigma es que "sobra" un argumento en los subprogramas (el propio objeto). En Python, al construir la plantilla de un método, el primer argumento es siempre *self*. A programadores experimentados en OO, puede parecerles un artefacto poco elegante, pero ciertamente ayuda a los nóveles a entender quién hace qué.

13. Conclusiones

El entusiasmo que se percibe en los programadores del lenguaje Python y en la documentación que ellos generan, ya de por sí, suponen un claro incentivo a acercarnos al lenguaje: ¿qué tendrá de diferencial?

En efecto, en algunos aspectos es completamente diferente de sus predecesores. Resalto dos: La relevancia que se concede a la legibilidad y mantenibilidad y su peculiar gestión de la memoria, que conlleva consecuencias en el comportamiento de las variables, tanto en la asignación, como en el paso de parámetros a los subprogramas. Esto último puede resultar no trivial a un programador, incluso experto en otros lenguajes.

Mi objetivo era facilitar el acercamiento al lenguaje Python, recorriendo (en no más de 7 folios) el subconjunto de constructores requerido para abordar la programación en el contexto del paradigma procedimental estructurado y modular; reutilizando lo conocido a través de otros lenguajes y deteniéndonos en los aspectos diferenciadores de éste. Adelantándonos a posibles quebraderos de cabeza. No he conseguido la brevedad deseada, pero la Figura 5 proporciona una visión global de este subconjunto.

Python 3.4.1: manejo excepciones.py - C:/Python34/manejo excepciones.py

```python
seguir='si'
while seguir=="si":
    try:
        dividendo = int(input('Introduce el dividendo: '))
        divisor = int(input('Introduce el divisor: '))
        print('El cociente de la división de ambos números es', dividendo//divisor)
    except ZeroDivisionError:
         print ('la división por 0 provoca infinito')
    except: print('cadena de caracteres no es divisible')
    seguir=input('deseas volver a intentarlo (si/no): ')
```

Python 3.4.1 Shell

```
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:
D64)] on win32
Type "copyright", "credits" or "license()" for more
>>> ================================ RESTART ======
>>>
Introduce el dividendo: 6
Introduce el divisor: 3
El cociente de la división de ambos números es 2
deseas volver a intentarlo (si/no): si
Introduce el dividendo: 5
Introduce el divisor: 4
El cociente de la división de ambos números es 1
deseas volver a intentarlo (si/no): si
Introduce el dividendo: 7
Introduce el divisor: 0
la división por 0 provoca infinito
deseas volver a intentarlo (si/no): si
Introduce el dividendo: caca
cadena de caracteres no es divisible
deseas volver a intentarlo (si/no): no
>>>
```

Figura 4: Manejo de excepciones en Python.

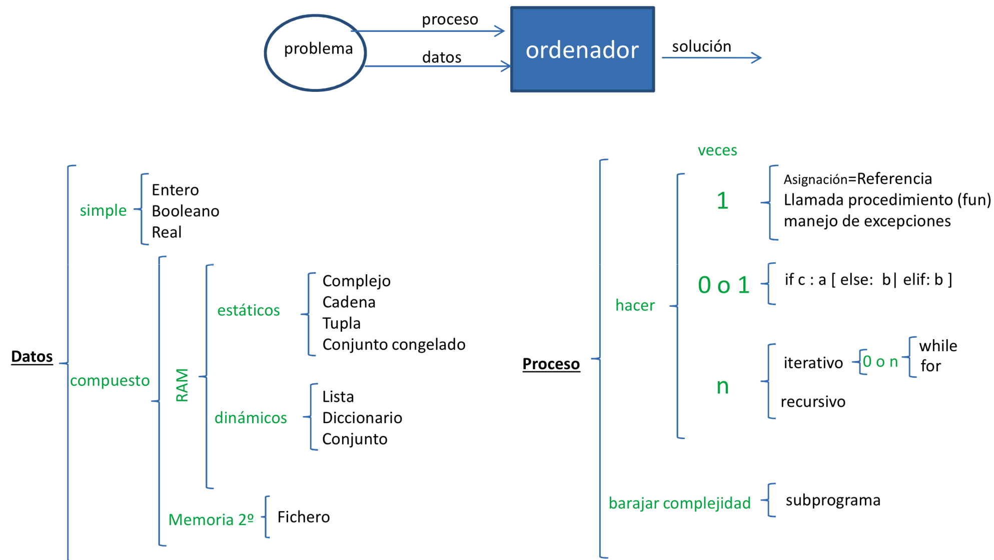


Figura 5: Programación estructurada en Python.

Referencias

[1] Philip Guo: *Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities.* Disponible en `http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext`. Fecha de último acceso: Septiembre de 2014.

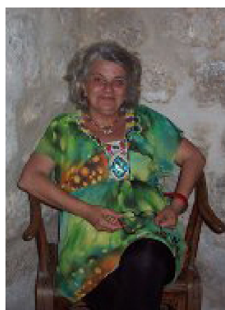
[2] Javier Montero: *El club del autodidacta.* Sección de Python. Blog disponible en `http://elclubdelautodidacta.es/wp/indice-python/` Fecha de último acceso: Septiembre de 2014.

[3] Ariel Ortiz: *EduPython. Impulsando el uso del lenguaje Python en la educación.* Blog disponible en `http://edupython.blogspot.com.es/2013/08/donde-quedo-el-do-while.html` Fecha de último acceso: Septiembre de 2014.

[4] Python Software Foundation: *Brief tour of the standard library.* Disponible en `https://docs.python.org/3/tutorial/stdlib.html` Fecha de último acceso: Septiembre de 2014.

[5] Python Software Foundation: *Python Success Stories*. Disponible en `https://www.python.org/about/success` Fecha de último acceso: Septiembre de 2014.

[6] Python Software Foundation: *The Python Standard Library. Built-in Functions.* Disponible en `https://docs.python.org/3.3/library/functions.html`. Fecha de último acceso: Septiembre de 2014.



Rosalía Peña: Estudiando tercero de Químicas en la Universidad de Murcia, inició su experiencia laboral, como por casualidad, en el Centro de Proceso de Datos en 1977, donde continúo hasta terminar su doctorado en Ciencias en el 1984. Desde entonces ha ido saltando del ejercicio profesional en empresa a la docencia, compatibilizándolo a veces y completando diversos estudios.
Titular de Escuela en el Departamento de Informática de la Universidad de Murcia hasta el 1988. Trabajó durante de 9 años en el Departamento de Nuevas Tecnologías de ERIA y en ERITEL en el Departamento de Multimedia. Ha sido profesora en la Universidad Europea de Madrid, actualmente es titular en la Universidad de Alcalá.
Master en Bioquímica clínica por la Universidad Complutense en 1994 e Ingeniero en Informática de Gestión por la Universidad Carlos III de Madrid en 2006.
Es autora de libros sobre Bases de datos, Protección de la Información, y Gestión de la información digital, así como de más de 25 ponencias y 15 artículos en revistas nacionales e internacionales. Ha colaborado en más de 40 cursos de formación continuada en informática para adultos.
Siempre interesada en la pedagogía y la formación integral del individuo es miembro fundador de AENUI (Asociación de Enseñantes Universitarios de Informática) y ha estado 10 años en el comité de Programa de JENUI. Premio AENUI a la calidad docente en 2014.

2015 R. Peña. Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional que permite copiar, distribuir y comunicar públicamente la obra en cualquier medio, sólido o electrónico, siempre que se acrediten a los autores y fuentes originales y no se haga un uso comercial.