Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

7 de noviembre de 2024

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Licencia: Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2024
Web: http://pyciencia.taniquetil.com.ar/

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

VERSIÓN PRELIMINAR

Índice general

Prefacio 2

Índice general 3

I Python 4

1. Controles de flujo 5

1.1. If, elif, else . . . 5

1.2. While . . . 8

1.3. For . . . 10

1.4. Excepciones . . . 15

1.4.1. ¿Qué son las excepciones? . . . 15

1.4.2. Capturando y levantando excepciones . . . 19

II Herramientas fundamentales 25

III Temas específicos 26

IV Apéndices 27

A. Zen de Python 28

Bibliografía 29

VERSIÓN PRELIMINAR

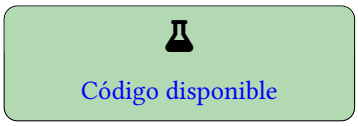
Parte I
Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

1 | Controles de flujo

En este capítulo mostraremos las distintas declaraciones de Python que nos permiten controlar el flujo de ejecución de un programa, que hasta ahora veíamos como lineal.



Python tiene pocas estructuras en este sentido (no repitiendo innecesariamente funcionalidad, "simple es mejor que complejo"), pero las pocas que tiene son poderosas e interesantes.

En este capítulo también incluiremos *excepciones*, que aunque también se utilizan para manejar errores, en realidad es un concepto más amplio (se utilizan para situaciones excepcionales, no sólo errores), y la naturaleza de las mismas hacen que el flujo del programa se modifique.

1.1. If, elif, else

Comenzamos con el control de flujo más sencillo, el `if`. La estructura básica es muy simple:

```
if <expresión>:
    <bloque de código>
```

Si la expresión es verdadera, se ejecuta el bloque de código; si no, no.

¿Qué es una "expresión" en este contexto? Una expresión es una combinación de valores, variables, operadores y llamadas a funciones. Cuando sea evaluada, para el `if` va a terminar siendo falsa o verdadera, no hay otra.

Más abajo seguiremos hablando de las expresiones, pero ahora centrémonos en el bloque de código que es lo que se ejecuta si la expresión es verdadera. No es más que una secuencia de líneas de código, que puede ser una o más (no cero, y sin límites prácticos en su cantidad). El comienzo y finalización del bloque está marcado por su sangría (que muchas veces llamamos "indentación", término que aunque es usado muchísimo es solamente un anglicismo de *indentation*).

Esa sangría puede ser cualquier cantidad de espacios o tabuladores (¡pero no mezclarlos! y se recomienda usar cuatro espacios), con la condición de que sea consistente. Por ejemplo, si un bloque arranca con la linea cuatro espacios a la derecha, siempre estará sangrado igual durante todo el bloque (a menos que haya bloques anidados, claro) y luego terminará volviendo a la columna original.

Ejemplo:

```
if foo == 42:
    print("foo vale 42")  # arranca el bloque con 4 a la derecha
    print("segunda linea")  # segunda linea ok
      print("tercera")  # esta está mal, porque está demasiado a la derecha
    if bar == 33:
        print("bar es 33")  # esta está bien, es un bloque nuevo adentro del otro
    print("muchas líneas")  # ok también: cierra el bloque interior, y vuelve a la columna original
  print("vamos cerrando")  # mal! si cerramos el bloque debemos volver a la columna original de este
print("afuera")  # esta está bien, ya afuera del bloque del if
```

A la estructura del **if** le podemos agregar una especie de continuación:

```
if <expresión 1>:
    <bloque de código 1>
elif <expresión 2>:
    <bloque de código 2>
```

Si la expresión 1 es verdadera, se ejecutará el bloque de código 1 y se ignorará el resto. Pero si la expresión 1 es falsa, se evalúa la expresión 2: si es verdadera se ejecutará el bloque 2, sino termina.

Y finalmente podemos agregarle una especie de salida final:

```
if <expresión 1>:
    <bloque de código 1>
elif <expresión 2>:
    <bloque de código 2>
else:
    <bloque de código 3>
```

El bloque de código 3 se ejecutará si todas las expresiones de la estructura fueron evaluadas a falso.

El **if** es obviamente obligatorio (arranca la estructura), pero el **elif** es opcional (y se pueden poner cuantos queramos, uno abajo del otro cada uno con su expresión a evaluar), y el **else** es también opcional pero no puede haber más de uno.

Veamos un ejemplo más real:

CELL 01

```
from datetime import date

hoy = date.today()
```

Cell 02

```
if hoy >= date(2030, 1, 1):
    print("El futuro ya llegó")
elif date(2020, 1, 1) <= hoy < date(2030, 1, 1):
    print('La famosa "década del 20"')
else:
    print("El lejano pasado")
```

```
La famosa "década del 20"
```

Claro que esa estructura se puede extender mucho más. En la actualidad Python no tiene una declaración `case` como muchos otros lenguajes. Esto se resuelve con estructuras **`if`**/**`elif`** si son relativamente pocos casos, o guardando funciones en un diccionario si son más.

En 2021, en la versión 3.10, Python ganó la posibilidad de realizar *pattern matching*, una característica bastante útil con muchas vueltas interesantes. Con esta nueva funcionalidad, usándola de forma más bien básica, se podría tener algo similar a la declaración `case`, pero les recomendamos mirar su PEP [2] para profundizar sobre este tema.

Volvamos sobre algo que prudentemente esquivamos al principio de la sección: las expresiones.

Decíamos que una expresión es una combinación de valores, variables, operadores y llamadas a funciones. Esto es bastante genérico, y hay pocas cosas que no pueden ser incluidas en una expresión, como definiciones de funciones o clases, importar módulos, etc. En la práctica podemos hacer casi todo lo que deseamos, y esto nos permite ser bastante expresivos, por ejemplo como veíamos arriba al comparar un valor con el resultado de la función `date` que estamos llamando ahí mismo.

A esto hay que sumarle que todos los tipos de datos integrados en el lenguaje también son evaluables a verdadero o falso, lo cual podemos revisar sencillamente con la función `bool` (como regla general, el objeto evalúa a False si vale cero o está vacío, sino a True).

Cell 03

```
bool(-1)
```

```
True
```

Cell 04

```
bool([])
```

```
False
```

Cell 05

```
bool(None)
```

```
False
```

Cell 06

```
bool("")
```

```
False
```

Cell 07

```
bool("falso")
```

```
True
```

Y además tenemos los operadores de comparación:

- == igual a
- != diferente de
- **is** es el mismo objeto que (identidad)
- < menor que
- <= menor o igual que
- > mayor que
- >= mayor o igual que

Todo esto nos permite ser muy expresivos al armar las estructuras **if**.

1.2. While

El **while** es una estructura de control de flujo que nos arma un bucle alrededor de ese bloque de código, repitiendo ese bloque en función de la evaluación de una expresión:

```
while <expresión>:
    <bloque de código>
```

Al arrancar el bucle, Python evalúa la expresión, si es falsa sale y nunca ejecuta el bloque de código. Si es verdadera ejecuta ese bloque de código, y vuelve a evaluar la expresión, si es falsa sale, sino ejecuta el bloque, y así hasta que la expresión de falso o se interrumpa por algo (pero potencialmente durante mucho mucho tiempo, hasta que el Sol se apague, digamos).

Veamos un ejemplo:

Cell 01

```
a = 0
while a < 3:
    a += 1
    print(a)
```

```
1
2
3
```

Ya hablamos al explicar el **if** 1.1 tanto de la expresión como del bloque de código, no hay mucho para agregar en ese aspecto. Por otro lado, con el **while** vemos que aparecen tres nuevas declaraciones, veámoslas.

El **break** nos permite interrumpir el bucle en la mitad de su ejecución. Si el código toca un **break**, entonces, el bucle se corta y sigue con lo que venía a continuación del mismo, sin terminar el bucle y sin volver a evaluar la expresión.

CELL 02

```
a = 0
while a < 5:
    a += 1
    if a == 3:
        break
    print(a)
```

```
1
2
```

El **continue** nos permite abortar la pasada actual del bucle, volviendo al principio del mismo, lo que incluye volver a evaluar la expresión. Vemos a continuación como el "3" no se imprime, porque al tocar el **continue** vuelve a recomenzar el bucle, sin llegar al `print` en ese caso.

CELL 03

```
a = 0
while a < 5:
    a += 1
    if a == 3:
        continue
    print(a)
```

```
1
2
4
5
```

Si usamos el **else** vamos a poder decidir si el bucle **while** terminó porque su expresión evaluó a falso o fue cortado con un **break**. Lo podemos pensar como en el **if**: si la expresión evalúa a verdadero se ejecuta el bucle, si evalúa a falso se ejecuta el bloque del **else**.

La combinación **while** con el **else** no es muy utilizado, en parte porque no lo vemos en otros lenguajes, pero es muy útil cuando justamente queremos saber si salimos del while en un caso o en el otro (lo que se resuelve en otros lenguajes utilizando otra variable como bandera).

Veamos ambos comportamientos en los siguientes ejemplos:

CELL 04

```
a = 0
while a < 2:
    a += 1
    if a == 3:
        break  # realmente no va a llegar acá
    print(a)
else:
    print("en el else")
print("afuera")
```

```
1
2
en el else
afuera
```

CELL 05

```
a = 0
while a < 4:  # ahora sí vamos a tocar el break
    a += 1
    if a == 3:
        break
    print(a)
else:
    print("en el else")
print("afuera")
```

```
1
2
afuera
```

1.3. For

El **for** es una declaración que nos permite recorrer iterables, ejecutando un bloque de código por cada uno de esos iterables (y haciendo referencia al elemento obtenido del iterable en cada momento usando un nombre que nosotros especificamos).

Si entendemos esa definición, la estructura es casi autodescriptiva:

```
for <nombre> in <iterable>:
    <bloque de código>
```

Veamos un ejemplo sencillo:

CELL 01

```
nros = [1, 2, 3]
for n in nros:
    print(n)
```

```
1
2
3
```

En realidad en lugar del "nombre" podemos tener varios nombres, si es que los elementos del iterable que recorremos pueden desempacarse correctamente, y obtendremos el mismo efecto que en la asignación múltiple (más específicamente, tenemos toda la experiencia del desempaquetado de tuplas que explicamos antes **??**):

CELL 02

```
cosas = [('a', 1), ('b', 2), ('c', 3)]
for letra, nro in cosas:
    print(letra, nro ** 2)
```

```
a 1
b 4
c 9
```

Vale la pena que aclaremos que el **`for`** es más parecido al `foreach` de otros lenguajes, y diferente al `for` de C o al `do` de Fortran, por ejemplo, que sólo cuentan números entre un principio y final (en general para indizar una estructura y obtener los elementos internos de la misma). En verdad la necesidad de trabajar con rangos de números es real, y para eso Python tiene una función integrada `range`, en el que podemos especificar el final del rango (arrancando por default en cero), o inicio y final, e incluso el paso:

CELL 03

```
list(range(5))
```

```
[0, 1, 2, 3, 4]
```

CELL 04

```
list(range(3, 8))
```

```
[3, 4, 5, 6, 7]
```

CELL 05

```
list(range(3, 8, 2))
```

```
[3, 5, 7]
```

CELL 06

```
list(range(3, 12, 2))
```

```
[3, 5, 7, 9, 11]
```

Prestemos atención que el "desde" es inclusivo, mientras que el "hasta" es exclusivo; esto aunque quizás sea sorprendente tiene sentido en la foto más grande del funcionamiento general de Python. Por ejemplo, si queremos los índices para una lista de 4 elementos, haremos `range(4)` y eso nos dará el 0, 1, 2 y 3 que son las posiciones de una lista de 4 elementos. También es muy práctica la propiedad de que si hacemos `range(M, N)` la cantidad de números que obtendremos es `N - M`.

Para el ejemplo utilizamos el `list` porque el `range` es un generador de números, entonces tenemos que consumirlo para ver esos números. Por supuesto que lo podemos iterar directamente con el **`for`**:

CELL 07

```
for n in range(2, 5):
    print("El cuadrado de {} es {}".format(n, n ** 2))
```

```
El cuadrado de 2 es 4
El cuadrado de 3 es 9
El cuadrado de 4 es 16
```

El **`for`** es un bucle, y como con el otro bucle de Python (el **`while`**, de la Sección 1.2) podemos afectar el comportamiento utilizando el **`break`** (para interrumpir el bucle y salir), el **`continue`** (para abortar la pasada del bucle y volver al principio, obteniendo un nuevo elemento del iterable), e incluso el **`else`** (para discernir si el **`for`** terminó porque se nos acabó el iterable o porque cortamos con un **`break`**.

Veamos algunos ejemplos usando estas funcionalidades. Arrancamos con el **`break`** y el **`continue`**, viendo como corta en un caso y como vuelve al principio en el otro esquivando el resto del bucle:

CELL 08

```
números = [1, 2, 3, 4]
for n in números:
    if n == 3:
        break
    print(n)
```

```
1
2
```

Cell 09

```
for n in números:
    if n == 3:
        continue
    print(n)
```

```
1
2
4
```

Y veamos el **else**, para el caso en que terminamos el **for** por consumir totalmente el iterable o porque encontramos un **break**:

Cell 10

```
for n in números:
    print(n)
else:
    print("se nos terminó el iterable")
```

```
1
2
3
4
se nos terminó el iterable
```

Cell 11

```
for n in números:
    if n == 2:
        break
    print(n)
else:
    print("se nos terminó el iterable")
```

```
1
```

Hay un caso de uso típico en los **for** que es arrancar con una lista, realizarle una operación, y terminar con otra lista con los resultados de esa operación. Por ejemplo, podemos tener una lista de números y queremos calcular sus cuadrados:

Cell 12

```
números = [1, 3, -2, 2, 0, 5]
cuadrados = []
for n in números:
    cuadrados.append(n ** 2)

cuadrados
```

```
[1, 9, 4, 4, 0, 25]
```

Esta construcción es tan usual que Python tiene una sintaxis especial que nos permite escribir lo mismo pero de forma más reducida, con la ventaja que hasta queda más legible; se denomina "comprensión de listas" (en inglés *list comprehension*), y se define usando corchetes para delimitar

la estructura, con los elementos sintácticos del `for` adentro:

Cell 13

```
números = [1, 3, -2, 2, 0, 5]
cuadrados = [n ** 2 for n in números]
cuadrados
```

```
[1, 9, 4, 4, 0, 25]
```

Podemos leer la segunda linea como "armamos una lista con ene al cuadrado para cada ene en números", y hace exactamente eso. Es mucho más fácil de entender que el `for` del ejemplo anterior que hace lo mismo pero a lo largo de varias líneas (entonces lo tenemos que seguir, ir y volver con la vista, entender qué hace con la lista que definimos al principio, etc.); en el caso de la "comprensión de listas" al primer vistazo (cuando reconocemos la estructura) ya sabemos que generamos una nueva lista realizando una operación con los elementos de un iterable, y nada más.

En realidad podemos complejizar apenas esa estructura, para el caso en que queramos filtrar algunos elementos del iterable fuente. Veamos los mismos ejemplos que recién pero calculando logaritmos solamente para los valores mayores a cero.

Cell 14

```
import math
números = [1, 3, -2, 2, 0, 5]
logs1 = []
for n in números:
    if n > 0:
        logs1.append(math.log(n))

logs1
```

```
[0.0, 1.0986122886681098, 0.6931471805599453, 1.6094379124341003]
```

Cell 15

```
logs2 = [math.log(n) for n in números if n > 0]
logs2
```

```
[0.0, 1.0986122886681098, 0.6931471805599453, 1.6094379124341003]
```

Una generalización de esta estructura nos permite armar una "comprensión de conjuntos" (al delimitar la estructura con llaves) e incluso una "comprensión de diccionarios" (al delimitar la estructura con llaves y tener clave y valor separados por dos puntos:

Cell 16

```
números = [1, 3, -2, 2, 0, 5]
[n ** 2 for n in números]
```

```
[1, 9, 4, 4, 0, 25]
```

Cell 17

```
{n ** 2 for n in números}
```

```
{0, 1, 4, 9, 25}
```

Cell 18

```
{n: n ** 2 for n in números}
```

```
{1: 1, 3: 9, -2: 4, 2: 4, 0: 0, 5: 25}
```

1.4. Excepciones

El sistema de manejo de errores de Python es a través de excepciones.

1.4.1. ¿Qué son las excepciones?

Una excepción es un evento que ocurre durante la ejecución normal del programa e interrumpe el flujo normal del programa. En general, cuando un programa en Python encuentra una situación que no puede manejar, levanta una excepción.

Entonces, a diferencia de otros lenguajes que luego de llamar a una función (por ejemplo) tenemos que revisar el resultado para ver si indica que hubo un problema, en Python el resultado será lo que tenga que devolver la función normalmente, si es que la función terminó sin inconvenientes. Pero si hubo un problema, obtendremos una excepción, que podemos manejar o dejar continuar.

Hagamos un pequeño programa para ver eso:

```
1 print("antes")
2 1 / 0
3 print("después")
```

Al ejecutar ese código, obtendremos:

```
$ python3 excep.py
antes
Traceback (most recent call last):
  File "x.py", line 2, in <module>
    1 / 0
ZeroDivisionError: division by zero
```

Vemos que se imprime el “antes”, pero luego la ejecución se interrumpió al encontrar un error (al intentar dividir por cero). En ese punto, se levantó una excepción, y como no se capturó la terminó agarrando el intérprete de Python, el que interrumpió al programa y mostró un *traceback* por pantalla. Un traceback (término que usamos en inglés, ya que nunca se popularizó decirles “trazas de rastreo”) es información que nos da el intérprete para entender de dónde viene el problema.

Tiene tres partes, un título que nos indica el comienzo del traceback (linea 3), un contenido cuyo largo dependerá de cuan profundo en la pila de llamadas a funciones haya sucedido el problema (líneas 4 y 5), y una última linea (la 6) mostrando el tipo de excepción y un mensaje (en nuestro caso `ZeroDivisionError`, y el mensaje indicando eso).

Para entender mejor la parte del medio veamos un ejemplo apenas más complejo (usando funciones, aunque nos adelantemos un poco a cuando las expliquemos formalmente en la Sección ??).

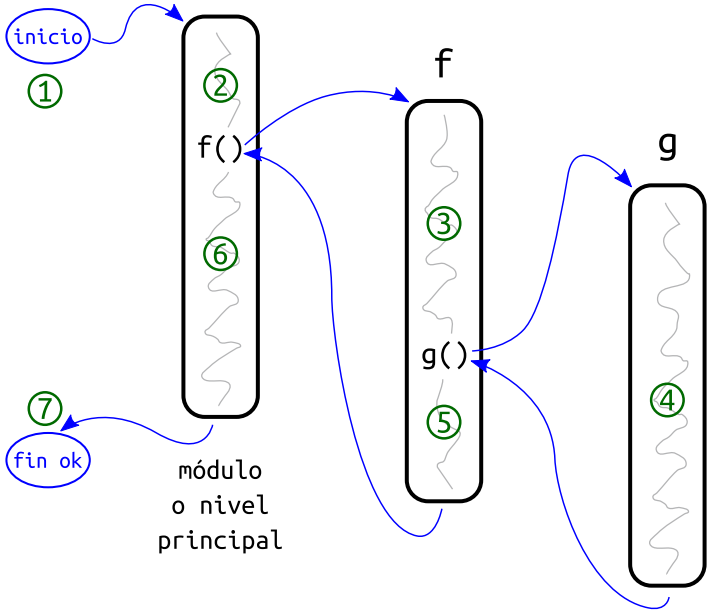
Veamos primero que pasaría si todo sale bien y no nos encontramos con ningún problema:

```
1  print("1. inicio")
2
3  def g():
4      print("4. en g")
5
6  def f():
7      print("3. antes en f")
8      g()
9      print("5. después en f")
10
11 print("2. antes en el módulo")
12 f()
13 print("6. después en el módulo")
14
15 print("7. fin")
```

Al ejecutar ese código, obtendremos:

```
1. inicio
2. antes en el módulo
3. antes en f
4. en g
5. después en f
6. después en el módulo
7. fin
```

Los números se corresponden al siguiente diagrama donde vemos la secuencia de ejecución de las distintas funciones:



Ahora veamos qué pasaría si en la función "g" nos encontramos con un problema. Modifiquemos esa función:

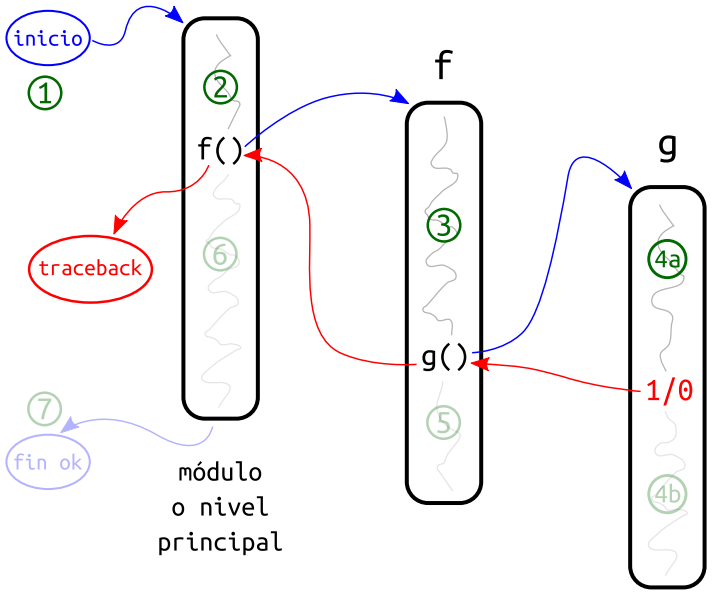
```
1  print("1. inicio")
2
3  def g():
4      print("4a. antes en g")
5      print("¿uno sobre cero?", 1 / 0)
6      print("4b. después en g")
7
8  def f():
9      print("3. antes en f")
10     g()
11     print("5. después en f")
12
13 print("2. antes en el módulo")
14 f()
15 print("6. después en el módulo")
16
17 print("7. fin")
```

Vemos que tenemos los resultados de los primeros prints y luego un traceback, pero no lo que veíamos antes:

```
1. inicio
2. antes en el módulo
3. antes en f
4a. antes en g
Traceback (most recent call last):
  File "x.py", line 14, in <module>
```

```
    f()
  File "x.py", line 10, in f
    g()
  File "x.py", line 5, in g
    print("¿uno sobre cero?", 1 / 0)
ZeroDivisionError: division by zero
```

Podemos ver mejor el flujo de ejecución en este otro diagrama que representa lo que acabamos de experimentar:



Allí vemos que el flujo de ejecución se modifica. Cuando llegamos al punto del problema, el resto de la función "g" no se ejecuta, sino que se genera una excepción que vuela hasta el punto donde la función fue llamada (linea roja que llega a "f"). El resto de esta función tampoco se ejecuta, como la excepción no fue capturada seguirá subiendo por las funciones hasta llegar a nivel de módulo, donde Python interrumpe el proceso mostrando el traceback.

La mejor forma que tenemos para entender lo que está pasando cuando hay un error es leer cuidadosamente el traceback. Recomendamos hacerlo de abajo para arriba.

Para el ejemplo anterior, vemos que tuvimos una excepción por división por cero en la linea 5 en la función "g" (donde hicimos `1 / 0`), lo cual viene de la linea 10 en la función "f" (donde hicimos `g()`), que a su vez viene de la linea 14 a nivel del módulo (donde hicimos `f()`).

Como tenemos un stack de largo tres (el cuerpo principal del programa más las dos funciones) tenemos tres pares de líneas en el centro del traceback.

Entonces, cada par de líneas de "la parte central" del traceback corresponde a un nivel del stack, y en cada caso nos muestra dónde se sucedió el problema para ese nivel del stack (en qué archivo, en qué linea y el contexto), y luego la linea en cuestión (que podríamos ver en el archivo/posición indicado, pero así es más cómodo).

1.4.2. Capturando y levantando excepciones

Podemos capturar las excepciones que se sucedan con un bloque **try**. La construcción típica es usarlo en conjunto al **except**:

```
try:
    <bloque de código>
except:
    <bloque de código>
```

El primer bloque de código, correspondiente al **try** es el supervisado, si se sucede alguna excepción en ese bloque, el **except** entra en juego y se ejecutará el segundo bloque de código; si ninguna excepción se levanta en el bloque supervisado, el **except** será ignorado completamente.

Veamos ambas situaciones en un ejemplo mínimo:

Cell 01

```
try:
    print("uno sobre dos:", 1 / 2)
except:
    print("hubo un problema")
```

Cell 02

```
try:
    print("uno sobre cero:", 1 / 0)
except:
    print("hubo un problema")
```

```
hubo un problema
```

Hay dos reglas de oro para seguir cuando escribimos estas estructuras. La primera es que debemos supervisar el mínimo de código posible (minimizar la cantidad de código dentro del bloque del **try**), la segunda es que debemos capturar solamente las excepciones que estamos esperando que puedan suceder (especificar el **except** lo más posible, no como hasta ahora que está capturando todo).

La primer regla es fácil de entender, pero veamos un ejemplo de cómo es útil especificar el **except** lo más posible. Supongamos el siguiente código, donde obtenemos un valor (supongamos de una medición) y calculamos uno sobre eso; excepcionalmente podemos tener un cero como valor, pero eso sería que el instrumento está descalibrado, entonces lo informamos y listo:

Cell 03

```
valor = 3  # todo bien
try:
    print("uno sobre algo:", 1 / valor)
except:
    print("instrumento descalibrado")
```

```
uno sobre algo: 0.3333333333333333
```

CELL 04

```
valor = 0  # efectivamente descalibrado
try:
    print("uno sobre algo:", 1 / valor)
except:
    print("instrumento descalibrado")
```

```
instrumento descalibrado
```

Sin embargo, supongamos que tenemos un problema más serio, y por un error en nuestro programa terminamos teniendo otra cosa como valor, una cadena:

CELL 05

```
valor = "error"  # otro problema desconocido
try:
    print("uno sobre algo:", 1 / valor)
except:
    print("instrumento descalibrado")
```

```
instrumento descalibrado
```

El mensaje que estamos dando es totalmente equivocado. Tengamos en cuenta que mucha veces también en el bloque del **`except`** se toman acciones para corregir o paliar el problema, y si tomamos las acciones equivocadas podemos estar complicando aún más la situación.

Entonces, tenemos que ser lo más específicos posibles al capturar la excepción. Para el ejemplo que estamos viendo, nosotros sabemos que podemos llegar a tener una **`ZeroDivisionError`**, ¡entonces capturemos sólo eso! Si le especificamos un tipo de excepción al **`except`** (o más de uno, entre paréntesis), capturará la excepción y ejecutará el bloque de código sólo si la excepción es de ese tipo (o de algunos de los varios tipos que pusimos entre paréntesis):

CELL 06

```
valor = 0
try:
    print("uno sobre algo:", 1 / valor)
except ZeroDivisionError:
    print("instrumento descalibrado")
```

```
instrumento descalibrado
```

CELL 07

```
valor = "error"
try:
    print("uno sobre algo:", 1 / valor)
except ZeroDivisionError:
    print("instrumento descalibrado")
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-7-3bd734f41923> in <module>
      1 valor = "error"
      2 try:
----> 3     print("uno sobre algo:", 1 / valor)
      4 except ZeroDivisionError:
      5     print("instrumento descalibrado")

TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Entonces, en el primer caso tenemos el mensaje que esperábamos, mientras que en el segundo caso la excepción no es capturada, la termina agarrando Python y nos genera el traceback correspondiente, lo cual está perfectamente bien, porque es información útil para encontrar el error en nuestro programa.

Hay situaciones, sin embargo, en que igualmente queremos capturar todo lo que pueda llegar a suceder, lo cual se justifica en procesos que son muy largos y que no queremos que se interrumpan nunca, o siempre tienen que estar levantados, y necesitamos capturar todas las excepciones para poder informarlas y seguir trabajando.

Pero incluso en estas situaciones no es recomendable poner el `except` "pelado", ya que hay excepciones que son internas al funcionamiento de Python y no debemos capturarlas. En estos casos es muy útil que las excepciones en Python estén dispuestas en forma de árbol, y que especificando un tipo de excepción realmente estamos capturando las excepciones de ese tipo y todas las de su rama (este árbol lo podemos ver en la documentación [3].

Entonces, podemos capturar `Exception` (que como vemos en el árbol de excepciones incluye a casi todas menos tres muy específicas), y obtendremos el efecto deseado. Es más, al especificar el tipo de excepción, podemos incluso ponerle un nombre a la excepción que capturamos y manejarla en el bloque de código.

Veamos todo esto en el ejemplo que traíamos, aprovechando la flexibilidad del `except` que nos permite especificar varios luego de un `try` (el comportamiento en estos casos es parecido a la cadena/secuencia que teníamos con los `if`/`elif`: se va verificando la excepción en orden en todos los `except`, si la excepción es capturada por uno de ellos se ejecuta su bloque de código y deja de verificarse en el resto.

CELL 08

```
valor = 0
try:
    print("uno sobre algo:", 1 / valor)
except ZeroDivisionError:
    print("instrumento descalibrado")
except Exception as exc:
    print("problema desconocido!", repr(exc))
```

```
instrumento descalibrado
```

CELL 09

```
valor = "queloqué"
try:
    print("uno sobre algo:", 1 / valor)
except ZeroDivisionError:
    print("instrumento descalibrado")
except Exception as exc:
    print("problema desconocido!", repr(exc))
```

```
problema desconocido! TypeError("unsupported operand type(s) for /: 'int' and 'str'")
```

En verdad, son más las herramientas que podemos usar con el **`try`**, no sólo tenemos el **`except`**, sino también el **`else`** y el **`finally`**, los cuales se pueden usar en cualquier combinación. La estructura completa sería:

```
try:
    <bloque de código>
except:
    <bloque de código>
else:
    <bloque de código>
finally:
    <bloque de código>
```

Veamos un resumen mostrando las características de cada una de esas partes:

- **`try`**: da comienzo al manejo de posibles excepciones, supervisando un bloque de código; es obligatorio incluirlo (da comienzo a la estructura) y puede estar una sola vez.
- **`except`**: ejecuta un bloque de código si en el código supervisado se levantó una excepción, y esa excepción es del tipo definido en el **`except`** (o incluida en su rama del árbol, como veíamos arriba); puede haber muchos o ninguno, si hay varios la comprobación de la excepción se hace en orden y se ejecuta el bloque de código solamente de aquel que captura la excepción.
- **`else`**: ejecuta su bloque de código solamente si en el código supervisado *no* se levantó una excepción; es opcional y puede haber a lo sumo uno.
- **`finally`**: ejecuta su bloque de código *siempre*, no importa qué haya pasado en el código supervisado; es opcional y puede haber a lo sumo uno.

Además de todo el manejo que podemos hacer sobre excepciones que son levantadas en alguna parte del código o por alguna situación, tenemos la opción de levantar nosotres mismos las excepciones integradas de Python, o incluso crear nuevas excepciones para nuestros programas.

Para levantar una excepción sólo tenemos que usar la declaración **raise** y la excepción y el mensaje que queremos levantar:

CELL 10

```
valor = -3
if valor < 0:
    raise ValueError("Valor inválido, no puede ser negativo.")
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-10-2d18b9ea9787> in <module>
      1 valor = -3
      2 if valor < 0:
----> 3     raise ValueError("Valor inválido, no puede ser negativo.")

ValueError: Valor inválido, no puede ser negativo.
```

Vemos en el ejemplo como el tipo de excepción y el mensaje usados son los mostrados en el traceback.

También podemos usar el **raise** sin especificar una excepción, pero solamente en el contexto de manejar alguna excepción que hayamos capturado: en este caso el **raise** lo que hará es "relevantar" la misma excepción.

Es muy útil para los casos donde capturamos una excepción pero solamente para manejar algunos casos y otros no. Por ejemplo, en las siguientes líneas capturamos un posible error que ignoramos en un caso y en otros no:

CELL 11

```
import os
to_remove = '/tmp/somefile.txt'
try:
    os.remove(to_remove)
except FileNotFoundError as err:
    if err.filename.startswith("/tmp/"):
        print("Ignoramos no haber encontrado un temporal")
    else:
        raise
```

```
Ignoramos no haber encontrado un temporal
```

CELL 12

```
to_remove = '/opt/somefile.txt'
try:
    os.remove(to_remove)
except FileNotFoundError as err:
    if err.filename.startswith("/tmp/"):
        print("Ignoramos no haber encontrado un temporal")
    else:
        raise
```

```
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-12-53dbdaf6d26c> in <module>
      1 to_remove = '/opt/somefile.txt'
      2 try:
----> 3     os.remove(to_remove)
      4 except FileNotFoundError as err:
      5     if err.filename.startswith("/tmp/"):

FileNotFoundError: [Errno 2] No such file or directory: '/opt/somefile.txt'
```

Definir nuestros propios tipos de excepciones es muy simple, aunque para ello necesitamos la sintaxis de clases (que veremos más adelante **??**). El único requisito es que tenemos que "heredar" una excepción integrada de Python; de nuevo, el concepto de "herencia" todavía no lo vimos, pero para nuestro propósito es lo que ponemos entre paréntesis en la definición, veamos un ejemplo simple:

CELL 13

```
class FueraDeRango(Exception):
    """El valor medido está fuera del rango permitido."""
```

Es importante elegir de qué excepción heredamos porque eso es lo que termina armando "el árbol de excepciones" que mencionábamos arriba cuando decíamos que **`except`** captura el tipo de excepción indicado y a toda su rama.

En el caso del ejemplo estamos heredando **`Exception`**, pero podríamos heredar alguna otra excepción cuya semántica esté más cerca de la excepción que estamos definiendo (para ello es interesante empaparse de los tipos de excepciones que trae Python [4]. Por ejemplo, en nuestro caso probablemente estaríamos mejor heredar de **`ValueError`**, ya que está relacionada con un valor obtenido.

Una vez definida, la usamos como cualquier otra excepción:

CELL 14

```
raise FueraDeRango("valor negativo")
```

```
FueraDeRango                              Traceback (most recent call last)
<ipython-input-14-5161da85a699> in <module>
----> 1 raise FueraDeRango("valor negativo")

FueraDeRango: valor negativo
```

Parte II
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

Parte III
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte II.

Parte IV
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [5].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

[1] URL: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

[2] URL: https://www.python.org/dev/peps/pep-0622/.

[3] URL: https://docs.python.org/dev/library/exceptions.html#exception-hierarchy.

[4] URL: https://docs.python.org/dev/library/exceptions.html#concrete-exceptions.

[5] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.

VERSIÓN PRELIMINAR