# 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





VERSIÓN PRELIMINAR

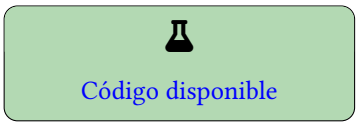
# Parte I
# Python

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 | Encapsulando código

Encapsular código es el acto que nos permite acomodar determinadas líneas de código en alguna estructura para poder reutilizarlas a conveniencia.




La forma más sencilla en Python de lograr esto son las funciones, que es lo primero que estudiaremos en este capítulo. Luego mostraremos las clases, que nos permiten encapsular no sólo el código sino también los objetos sobre los cuales trabaja ese código (abriéndonos las puertas a la Programación Orientada a Objetos), y finalmente hablaremos de módulos y paquetes, que son capas superiores que nos permiten encapsular funciones y clases para usarlas de distintos programas.

## 1.1. Funciones

Como mencionábamos al principio del capítulo, la función es la estructura más sencilla para encapsular código.

Nos permite escribir un bloque de código (con sus estructuras de control de flujo, con sus propios bloques de código, etc) de forma que después podremos ejecutar ese bloque de código "llamando" a la función desde cualquier lado.

La forma más sencilla de la estructura de una función es:

```
def <nombre>():
    <bloque de código>
```

Esa estructura, aunque funcional, no nos permite pasarle datos ni obtener un resultado. Pero nos sirve para empezar a familiarizarnos con las funciones. Tenemos entonces un "nombre" que es el que usaremos para identificar a la función, y un bloque de código (indentado, como corresponde).

Es importante entender la diferencia entre "definir" una función y "llamar a" (o "ejecutar") una función. En el primer caso solamente hacemos que Python compile la estructura y la tenga en memoria lista para usar, mientras que en el segundo caso es realmente cuando el bloque de código de la función se termina ejecutando.

En el siguiente ejemplo podemos ver primero la definición en sí de la función, y cómo podemos referenciarla con su nombre (en el `print`, o directamente en el intérprete interactivo, y la diferencia fundamental con ejecutar esa función, al final del ejemplo, cuando escribimos el

nombre de la función seguida de paréntesis (sin nada entre ellos, en este caso, porque la función no recibe parámetros).

CELL 01

```
def foo():
    print(5)
```

CELL 02

```
print(foo)
```

```
<function foo at 0x7fc314631af0>
```

CELL 03

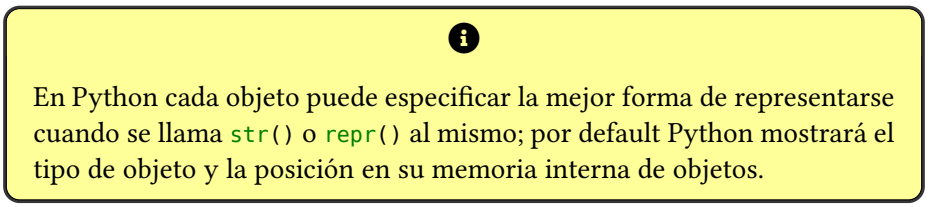
```
foo
```

```
<function __main__.foo()>
```

CELL 04

```
foo()
```

```
5
```



En Python cada objeto puede especificar la mejor forma de representarse cuando se llama `str()` o `repr()` al mismo; por default Python mostrará el tipo de objeto y la posición en su memoria interna de objetos.

Tener una función como esa es válido en algunos casos, pero en realidad la mayoría de las veces estaremos pasándole valores a la función y/o recibiendo resultados de la misma.

Para recibir valores, los tenemos que especificar en la definición de la función. Esto se logra de distintas maneras, y es bastante flexible (lo veremos más abajo), pero por ahora, simplificando, digamos que escribimos los nombres con los que haremos referencia a esos valores, y los podremos acceder desde el bloque de código de la función.

En el siguiente ejemplo definimos una función que recibe dos valores (sería un error pasarle uno o tres):

CELL 05

```
def foo(a, b):
    print(a, b)
```

CELL 06

```
foo
```

```
<function __main__.foo(a, b)>
```

CELL 07

```
foo(3, 4)
```

```
3 4
```

Hasta ahora la función ejecuta su bloque de código y termina. Por default, la función siempre devuelve `None` al terminar, pero tenemos control sobre eso mediante la declaración `return`.

Podemos poner cualquier cantidad de `return`s en una función. Si el flujo del código pasa por una linea con `return`, la función termina y devuelve lo que allí se indica (no importa si hay otros `return`s en otros lados de la función).

CELL 08

```
def foo(a, b):
    return a + b
```

CELL 09

```
r = foo(3, 4)
r
```

```
7
```

Prestemos atención al detalle de haber llamado a la función y realizar una asignación con el resultado de esa función, para poder trabajar luego con el mismo.

Tengamos en cuenta que podemos obtener más de un resultado cuando termina una función, para lo cual el `return` soporta que escribamos diferentes valores separados por coma y podemos acceder a esos valores con una asignación múltiple.

En realidad, a bajo nivel, lo que sucede es que el `return` está devolviendo una tupla con esos valores, y luego en la asignación del resultado entra en juego lo que llamamos *tuple unpacking* ??.

Ya con un ejemplo más complejo, armemos una función que recibe dos valores y devuelve la multiplicación y la división de ambos números (¡si es posible! si el segundo número es cero devuelve None allí):

CELL 10

```
def foo(a, b):
    mul = a * b
    if b == 0:
        return mul, None

    div = a / b
    return mul, div
```

Cell 11

```
foo(3, -2)
```

```
(-6, -1.5)
```

Cell 12

```
m, d = foo(3, -2)
```

Cell 13

```
m
```

```
-6
```

Cell 14

```
d
```

```
-1.5
```

Cell 15

```
print("Resultados: multip={} divis={}".format(m, d))
```

```
Resultados: multip=-6 divis=-1.5
```

Cell 16

```
m, d = foo(3, 0)
print("Resultados: multip={} divis={}".format(m, d))
```

```
Resultados: multip=0 divis=None
```

No hay más para explorar por el lado de devolver valores, así que volvamos sobre el otro lado de usar funciones: pasarle parámetros.

Vayamos mostrando las distintas alternativas. El modo más básico es lo que veníamos haciendo, definir algunos parámetros y pasar valores para los mismos. Veamos cuando esto funciona bien, y también los errores que tenemos al no respetar ese "acuerdo básico":

Cell 17

```
def foo(a, b):
    print(a, b)
```

Cell 18

```
foo(5, 6)
```

```
5 6
```

Cell 19

```
foo(5)
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-19-c5b1cea4a11b> in <module>
----> 1 foo(5)

TypeError: foo() missing 1 required positional argument: 'b'
```

Cell 20

```
foo(5, 6, 7)
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-20-5b675c25d8d5> in <module>
----> 1 foo(5, 6, 7)

TypeError: foo() takes 2 positional arguments but 3 were given
```

Un detalle básico pero interesante es que hay una correspondencia ordinal entre los parámetros especificados en la definición de la función y los valores que pasamos al llamarla: el `5` va a la `a` y el `6` va a la `b`; es por esto que llamamos "posicionales" a estos argumentos (lo vemos también en el mensaje de error en el ejemplo).

Cuando definimos la función podemos especificar que algunos de esos parámetros tengan un valor por default, entonces no va a ser necesario pasarlos cuando llamemos a la función:

Cell 21

```
def foo(a, b=1, c=3):
    print(a, b, c)
```

Cell 22

```
foo(9)
```

```
9 1 3
```

Cell 23

```
foo(9, 5)
```

```
9 5 3
```

Cell 24

```
foo(9, 5, 7)
```

```
9 5 7
```

En el ejemplo vemos que si pasamos sólo el valor para `a`, `c` y `b` tienen sus valores por default. En la segunda llamada pasamos valor para `a` y para `b` (de nuevo, porque son posicionales, el primer valor al primer parámetro, etc.), pero no para `c`. Y finalmente, vemos que si les pasamos valores para los tres, no se consideran sus valores por default.

¿Pero cómo haríamos en el ejemplo anterior para pasar un valor a `a` y a `c`, pero no a `b` (y que tome su valor por default)? Para ello nos tendríamos que salir del esquema de parámetros posicionales y empezar a nombrarlos, lo cual es tan sencillo como especificar para qué parámetro queremos que vaya cada valor:

Cell 21

```
def foo(a, b=1, c=3):
    print(a, b, c)
```

Cell 25

```
foo(9, c=7)
```

```
9 1 7
```

En este caso vemos que pasamos un `9` que va a `a` (¡posicional!) pero luego especificamos que el `7` va para `c`; a `b` no le terminamos pasando un valor, así que toma su default.

En realidad una vez que nombramos los parámetros, podemos escaparnos totalmente al orden de sus posiciones, más allá que en la definición tengan valores por default o no:

Cell 21

```
def foo(a, b=1, c=3):
    print(a, b, c)
```

Cell 26

```
foo(9, c=7, b=8)
```

```
9 8 7
```

Cell 27

```
foo(c=7, a=33)
```

```
33 1 7
```

Hasta ahora estamos manejando cantidad de fija de parámetros. Python soporta que en la definición usemos el `*` que consumirá todos aquellos valores que pasemos por posición que no hayan sido tomados todavía:

Cell 28

```
def foo(a, b, *c):
    print(a, b, c)
```

Cell 29

```
foo(1, 2)
```

```
1 2 ()
```

Cell 30

```
foo(1, 2, 3, 4)
```

```
1 2 (3, 4)
```

En el ejemplo vemos como en la primer llamada los dos valores que pasamos van a los primeros dos parámetros definidos, pero no quedó nada para c (entonces es una tupla vacía), mientras que en el segundo caso "sobraron" dos valores, entonces c si tiene contenido.

También Python nos ofrece el **, que de manera similar consumirá todos los nombrados que no hayan encontrado otro lugar:

Cell 31

```
def foo(a, b, **c):
    print(a, b, c)
```

Cell 32

```
foo(a=1, b=2)
```

```
1 2 {}
```

Cell 33

```
foo(a=1, c=7, b=3, d=8)
```

```
1 3 {'c': 7, 'd': 8}
```

Para el caso de * la estructura donde Python guarda los argumentos posicionales sobrantes es una tupla, ya que sólo es importante el orden, mientras que para el ** como tenemos valores y nombres, la estructura útil para guardar eso es el diccionario.

Obviamente se pueden combinar todos los casos que estuvimos viendo hasta recién .

Cell 34

```
def foo(a, b, *c, d=7, e=8, **f):
    print(a, b, c, d, e, f)
```

Cell 35

```
foo(1, 2)
```

```
1 2 () 7 8 {}
```

Cell 36

```
foo(1, 2, 3, 4, e=9, j=15)
```

```
1 2 (3, 4) 7 9 {'j': 15}
```

Todo esto funciona mientras no haya ambigüedades en la definición o en el llamado a la función; en esos casos Python mostrará un mensaje de error indicando el problema.

CELL 37

```
def foo(a=7, b):
    print(a, b)
```

```
  File "<ipython-input-37-258463965655>", line 1
    def foo(a=7, b):
            ^
SyntaxError: non-default argument follows default argument
```

CELL 38

```
def foo(a, b=7, **c):
    print(a, b, c)

foo(1, 3, a=5)
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-38-cf3d3e66d1d8> in <module>
      2     print(a, b, c)
      3
----> 4 foo(1, 3, a=5)

TypeError: foo() got multiple values for argument 'a'
```

Así como podemos usar en la definición el * para guardar excedentes posicionales en una tupla y ** para los excedentes nombrados en un diccionario, podemos usarlos en las llamadas a las funciones para "desarmar" una tupla con los valores o un diccionario con los nombres/valores:

CELL 39

```
def foo(a, b, c):
    print(a, b, c)
```

CELL 40

```
t = (1, 2, 3)
foo(*t)
```

```
1 2 3
```

CELL 41

```
d = {'b': 2, 'c': 3, 'a': 1}
foo(**d)
```

```
1 2 3
```

Cabe acotar que no estamos mencionando todos los casos posibles, y que hay más reglas y operadores (como forzar a que los parámetros sean nombrados o posicionales), y algunos detalles más, explicados en profundidad en la referencia del lenguaje [2].

### 1.1.1. Espacios de nombres

Cuando explicamos cómo funcionaba Python con sus objetos y nombres (en lugar de variables con valores, ver Sección **??**), usamos unos diagramas donde a la derecha teníamos los objetos en memoria, y a la izquierda otra zona donde poníamos los nombres. Este espacio reservado para los nombres se llama justamente "espacio de nombres" (en inglés *namespace*), y es una zona de memoria donde justamente se guardan los nombres que referencian a los otros objetos.

Traemos esto a colación en esta subsección porque en Python no tenemos solamente un espacio de nombres, sino que pueden haber muchos, y las funciones tienen mucho que ver en eso.

Cuando arranca Python tenemos un espacio de nombre que se conserva hasta que el proceso termina y es accesible desde todos lados: el espacio de nombre "global". Por otro lado, cada vez que ejecutamos una función, se crea otro espacio de nombres, "local" a la función, que permanecerá activo mientras las función se está ejecutando y desaparecerá cuando la misma termine.

Tenemos que tener en cuenta que lo que se destruye al terminar la función es el espacio de nombre, no los objetos referenciados por los mismos. Claro, algunos objetos quedarán sin referencia luego de que el espacio de nombre desaparezca (y de esos se encarga la administración automática de memoria de Python), pero puede ser que otros objetos estén referenciados de otros lados, y sigan vivos.

Veamos un ejemplo sencillo:

Cell 42

```
def foo(a, b):
    x = a * b
    return x
```

Cell 43

```
x = 3
y = 5
foo(x, y)
```

```
15
```

Cell 44

```
x
```

```
3
```

Analicémoslo en detalle, por partes. Lo primero que tenemos es una definición de una función (que todavía no ejecutamos, claro). Luego le ponemos nombre a dos enteros (`x` e `y`), que usamos para llamar a la función. En *ese* momento se ejecuta la función, que nos devuelve 15.

Si vamos a la ejecución de la función, vemos que esos dos enteros los recibe en dos parámetros que llama `a` y `b`, realiza un cálculo y devuelve ese valor. Como parte del procesamiento, la función también define un nombre `x`, pero este se define en el espacio de nombres *local* de la función (igual que `a` y `b`, para el caso).

La `x` definida en el espacio de nombres local apunta al entero 15, y luego de ejecutar la función vemos que, afuera, `x` sigue apuntando al 3 original. Esto es porque afuera estamos usando el

espacio de nombres global, no se nos mezcla con el espacio de nombres local de la función.

Es importante entender la diferencia entre "definir" un nombre en un espacio de nombres, y tener "acceso" a ese nombre (también decimos "ver" ese nombre, en inglés se usa *scope*). Veamos el siguiente ejemplo para resaltar esta diferencia:

CELL 45

```
def foo():
    y = 2
    z = 3
    print(x, y, z)
```

CELL 46

```
x = 8
y = 9
foo()
```

```
8 2 3
```

CELL 47

```
x
```

```
8
```

CELL 48

```
y
```

```
9
```

CELL 49

```
z
```

```
NameError                                 Traceback (most recent call last)
<ipython-input-49-3a710d2a84f8> in <module>
----> 1 z

NameError: name 'z' is not defined
```

Arrancamos definiendo una función (que veremos en detalle a continuación, cuando la ejecutemos), y luego se definen en el espacio de nombres global una `x` apuntando a un 8 y una `y` apuntando a un 9.

Cuando ejecutamos la función, esta primero define una `y` apuntando a un 2 y una `z` apuntando a un 3 (en ambos casos, en el espacio de nombres local de la función), y luego hace un print de tres nombres: para `x` e `y` es simple, porque está mostrando lo que encuentra en el espacio de nombres local, pero `z` nos puede sorprender.

Es aquí donde tenemos que entender que a nivel de visibilidad, desde adentro de la función Python intenta resolver el nombre primero buscando en el espacio de nombres local, y luego si no la encuentra allí busca en el espacio de nombres global. Esta secuencia es importante, porque eso determina que el `y` que encuentra es el que apunta al 2 (¡no al 9!), y para `z` que no está local

pero si global, igual la encuentra.

Por otro lado, desde afuera de la función no tenemos visibilidad a su espacio de nombres, por eso cuando al final queremos ver el valor de z nos da error de nombre, porque z no está definida en el espacio de nombres global (y es en el único en que busca).

Un detalle importante es que como desde adentro de la función tenemos visibilidad sobre los objetos del espacio de nombres global, si los objetos son mutables podremos modificarlos:

Cell 50

```
def foo():
    x.append(3)
```

Cell 51

```
x = [1]
x
```

```
[1]
```

Cell 52

```
foo()
x
```

```
[1, 3]
```

Cell 53

```
foo()
x
```

```
[1, 3, 3]
```

Aunque podría considerarse una mala práctica de programación (porque la función nos está cambiando objetos que viven fuera de ella), en algunos casos es útil y muy ventajoso poder hacer eso. ¡Usar con precaución!

### 1.1.2. Generadores

Los generadores son un tipo particular de objeto que cuando los iteramos nos dan elementos, pero no los tenían de antes. Los van generando en el momento. Un ejemplo integrado en Python es el `range`. Si hacemos `range(10 ** 100)` obtenemos un objeto que si le pedimos, nos dará enteros entre 0 y $10^{100}$ - 1, pero obviamente no preparó todos esos números en el momento. Los irá *generando*.

Se usan mucho en Python porque optimizan el uso de memoria y mejoran el rendimiento general. Veamos el siguiente ejemplo que aunque muy simple, muestra una optimización clara que repetida por varios rincones del lenguaje hacen una diferencia importante:

Cell 54

```
sum(range(10))
```

```
45
```

El ejemplo suma los números del 0 al 9, generados por el `range`. Si `range` (en vez de funcionar como generador) armara una lista con los números del 0 al 9, el efecto sería exactamente el mismo, con el detalle que como parte del proceso, se construyó una lista con todos los números en memoria simultáneamente, que luego fue consumida por el `sum`. Esa lista no sirvió para nada, en realidad, sólo ocupó memoria y tiempo para su creación/administración.

Hay distintas formas de construir generadores en Python, pero una de las más simples es armar una "función generadora". Parece una función normal con la excepción que dentro de su bloque de código usa la declaración **`yield`**, que justamente le cambia el comportamiento.

¿Recuerdan que dijimos que las funciones cuando las llamamos se ejecutan hasta que terminan, destruyendo su espacio de nombres local, y que cuando las volvemos a llamar vuelven a ejecutarse desde el principio? Bueno, justamente las funciones generadoras cambian ese comportamiento. Cuando llamamos a la función nos devuelve un generador. Cuando le vamos pidiendo elementos a ese generador lo que va a hacer es ejecutar esa función hasta que llega a un **`yield`**, devolviéndonos lo que allí se indica, "pausando" la ejecución de ese código. Y cuando le pidamos el próximo elemento al generador, ese código "se despertará" en el punto en que estaba y continuará su ejecución hasta que termine o encuentre otro **`yield`**.

Para ver esto en un ejemplo, primero hagamos una versión casera y recortada del `range` con una función clásica, para poder comparar ambos códigos.

Cell 55

```
def rango(limite):
    nros = []
    n = 0
    while n < limite:
        nros.append(n)
        n += 1
    return nros
```

Cell 56

```
rango(10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Cell 57

```
sum(rango(10))
```

```
45
```

En esta función vemos lo que mencionábamos arriba. La función genera una lista con los números, luego el `sum` la consume sumando esos números y produce el 45 como resultado. No sólo la lista es innecesaria, también mantener todos los números al mismo tiempo en memoria sólo para sumarlos es un desperdicio de recursos. Esto es obviamente un factor si tenemos una

lista muy grande, pero tampoco hay que desestimar la situación con estructuras pequeñas, porque cualquier necesidad de memoria puede disparar que el proceso que estamos ejecutando tenga que salir a pedirle memoria al sistema operativo, y eso siempre es caro.

Convirtamos la función de arriba en generadora.

CELL 58

```
def rango(limite):
    n = 0
    while n < limite:
        yield n
        n += 1
```

CELL 59

```
g = rango(10)
g
```

```
<generator object rango at 0x7fc314591d60>
```

CELL 60

```
next(g)
```

```
0
```

CELL 61

```
next(g)
```

```
1
```

CELL 62

```
sum(rango(10))
```

```
45
```

La estructura definida es muy similar, pero notemos que no tenemos la estructura interna `nros` (porque no estamos creando esa lista), y que aparece el famoso **`yield`**.

Cuando ejecutamos esta función generadora, realmente no se empieza a ejecutar el bloque de código, sino que obtenemos el objeto generador en sí, como mostramos en el ejemplo. Al hacer el primer `next` se empieza a ejecutar el código, hasta que llega al **`yield`**, allí devuelve el valor que teníamos en `n` (el `0`) y la ejecución del código queda suspendida (ya que el control lo tenemos nosotros en el intérprete interactivo). Cuando hacemos el segundo `next`, la ejecución no arranca desde el principio, sino que continua desde donde estaba, sumándole `1` a `n`, luego vuelve a evaluar la expresión del **`while`** y llega nuevamente al **`yield`**, devolviéndonos ahora el `1`.

Si siguiéramos pidiéndole números con el `next`, los seguiríamos obteniendo hasta que la expresión del **`while`** de falso, y en ese caso vemos que se termina la función. Como es una función generadora, cuando sale de la función en realidad se genera la excepción **`StopIteration`** que es la que usa Python para indicar que no hay más ítems para iterar. Esto nos permite integrar a estos generadores en todas las estructuras normales de Python.

Finalmente entonces, usamos en el `sum` la función generadora que armamos, que vemos que nos devuelve el mismo resultado que con la estructura clásica, pero sin construir la lista intermedia.

También podemos aplicar este concepto de "generador" a las comprensiones de listas ??, armando directamente "comprensiones generadoras", usando paréntesis en lugar de corchetes

Cell 63

```
(x ** 2 for x in range(3))
```

```
<generator object <genexpr> at 0x7fc314591e40>
```

Incluso podemos obviar esos paréntesis cuando tenemos a la expresión generadora dentro de una llamada a función (porque los dobles paréntesis son superfluos en Python).

Veamos un simple ejemplo donde sumamos los cuadrados del 0 al 9 de dos formas distintas.

Cell 64

```
sum([x ** 2 for x in range(10)])
```

```
285
```

Cell 65

```
sum(x ** 2 for x in range(10))
```

```
285
```

En ambos casos arrancamos con `range` (que es generadora), pero en el primero armamos una lista intermedia con los cuadrados (notemos los corchetes que arman la compresión de listas), mientras que en el segundo tenemos una comprensión generadora (no hace falta poner los paréntesis, ya que aprovechamos los de alrededor.

Para el `sum` es exactamente lo mismo, y tenemos el mismo resultado, pero el segundo caso es más rápido y hasta más legible y conciso.

## 1.2. Clases

Las "clases" son una forma de encapsular código junto a los objetos que son manejados por ese código.

Aunque suena simple, realmente esto nos permite separar en distintas estructuras los distintos objetos que necesitamos manejar con el código para procesarlos, lo cual nos permite modelar eficientemente la realidad que estamos tratando de representar, abriéndonos las puertas de la Programación Orientada a Objetos (en adelante "POO").

En este libro mostraremos el funcionamiento básico de las clases, la definición de su estructura, qué implica instanciarlas y las bases del funcionamiento de los objetos que obtenemos, pero no pretendemos enseñar POO en un capítulo, ya que es un tema para todo un libro (¡o más de uno!).

Entonces la idea es que podamos leer y entender código que usa clases, e incluso construir

algunas sencillas, sin pretender desbloquear todo su potencial.

La estructura básica de una clase es extremadamente simple:

```
class <nombre>:
    <bloque de código>
```

Con eso ya armamos una clase con el nombre que queramos, y tenemos un bloque de código para continuar. Como con las funciones, las clases nos crean un espacio de nombres diferente, y allí es que empezaremos a agregar código que luego utilizaremos más adelante.

Armemos un ejemplo con más elementos, para empezar a explicarlos:

CELL 01

```
class Foo:
    a = 3
    def f():
        print(8)

Foo
```

```
__main__.Foo
```

CELL 02

```
Foo.a
```

```
3
```

CELL 03

```
Foo.f
```

```
<function __main__.Foo.f()>
```

CELL 04

```
Foo.f()
```

```
8
```

En el bloque de código definimos una variable y una función, y luego las usamos. Veamos que hicimos `Foo.a` y `Foo.f()`, porque tanto "a" como "f" no están en el espacio de nombres global, sino en el de la clase (y como con los módulos, cuando hacíamos `math.sqrt(2)` ??, usamos el punto para indicar que estamos usando un nombre de "adentro" de otro objeto). Entonces, es claro que tanto "a" como "f" están adentro de la clase.

Lo que es raro en ese ejemplo es como estamos usando la clase. En realidad el uso normal de una clase es para generar objetos, como es regla en la POO. Es más, como ya estamos metiéndonos en ese mundillo, adoptemos dos términos que son de uso genérico allí y que vamos a encontrar en muchos textos. Las variables dentro de las clases y objetos las llamaremos "atributos", mientras que las funciones que definimos allí adentro las llamaremos "métodos". Entonces, de acuerdo con esta nueva terminología, diremos que tenemos "una clase Foo con un atributo a y un método f".

Ahora hagamos el último salto y armemos una clase que tiene sentido ser usada para instanciar objetos.

CELL 05

```
import math

class TriánguloRectángulo:
    def __init__(self, cateto1, cateto2):
        self.cateto1 = cateto1
        self.cateto2 = cateto2
    def hipotenusa(self):
        return math.sqrt(self.cateto1 ** 2 + self.cateto2 ** 2)
```

Epa, ¡cuantas cosas nuevas! Vayamos entendiéndolas por partes, viendo cómo encaja en este uso distinto que mencionamos arriba.

Las clases, en el paradigma de POO, son las estructuras que encapsulan el código para procesar determinada información, junto a dicha información. Y funcionan como plantillas, que nos darán distintos objetos cada vez que *instanciemos* la clase. Estos objetos serán del mismo tipo (el tipo de los objetos es la clase en sí), y por lo tanto el código encapsulado será el mismo, aunque ese código procesará la distinta información que tendremos en cada objeto.

En nuestro ejemplo tenemos una clase TriánguloRectángulo, donde encapsulamos un código (cómo calcular la hipotenusa a partir de los catetos) junto a la información en sí (los catetos). Si queremos trabajar con distintos triángulos, obviamente tendremos una variedad de pares de catetos, pero en todos los casos la forma de calcular la hipotenusa es la misma; esto está en linea con la filosofía de la POO de que estos objetos modelan y representan nuestra realidad.

Armemos entonces dos triángulos, para experimentar y seguir entendiendo ese código (sin repetir aquí la definición de la clase, por brevedad):

CELL 06

```
t1 = TriánguloRectángulo(4, 5)
t1
```

```
<__main__.TriánguloRectángulo at 0x7f18bc45f880>
```

CELL 07

```
t2 = TriánguloRectángulo(10, 1)
t2
```

```
<__main__.TriánguloRectángulo at 0x7f18bc45f910>
```

CELL 08

```
t1.cateto1
```

```
4
```

CELL 09

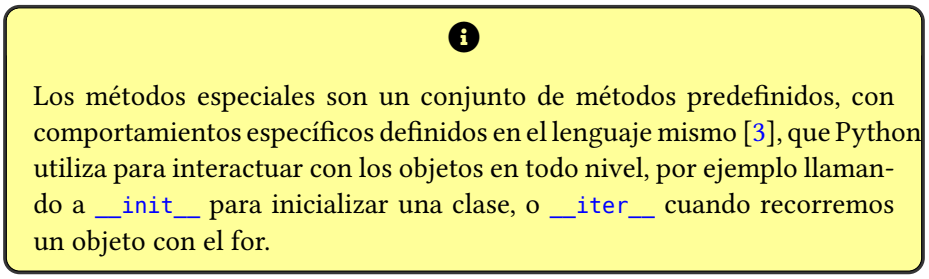
```
t2.cateto1
```

```
10
```

Vemos que al nombre de la clase le estamos agregando paréntesis, como hacemos con las funciones cuando las ejecutamos. Aquí es similar, pero a la clase la estamos *instanciando*, lo que nos devuelve un "objeto del tipo TriánguloRectángulo", que guardamos en t1 (y luego hacemos lo mismo con t2).

Cuando le decimos al intérprete interactivo que nos muestre esos objetos, vemos que nos dice que son del tipo TriánguloRectángulo y nos dice que están en posiciones de memoria distintas (con lo cual podemos deducir que son dos objetos distintos). Es más, cuando nos fijamos el valor de `cateto1` de ambos objetos vemos que obtenemos distintos valores (¡son distintos objetos!), cada uno teniendo el primer valor que pasamos cuando instanciamos la clase.

¿Cómo sucedió eso? Si volvemos a la definición de la clase, vemos que allí teníamos un método con un nombre especial, `__init__`. Este método se ejecuta automáticamente cuando instanciamos la clase donde está definido. Entonces, cuando hicimos `TriánguloRectángulo(4, 5)` se instanció la clase y se ejecutó ese método de inicialización, pasándole justamente estos valores que indicamos nosotros.



Los métodos especiales son un conjunto de métodos predefinidos, con comportamientos específicos definidos en el lenguaje mismo [3], que Python utiliza para interactuar con los objetos en todo nivel, por ejemplo llamando a `__init__` para inicializar una clase, o `__iter__` cuando recorremos un objeto con el for.

Pero si prestamos un poco más de atención veremos que en su definición `__init__` declara que tiene que recibir 3 parámetros (`self`, `cateto1` y `cateto2`), mientras que nosotros estamos pasando solamente dos. Es que para todo lo que es métodos en las clases, Python inserta automáticamente como primer parámetro al objeto mismo que estamos manejando, al que por convención llamamos `self`.

Y allí vemos que el cuerpo del método `__init__` lo que hace es crear los nombres `cateto1` y `cateto2` *adentro del objeto*, apuntando a los objetos recibidos. Entonces, cuando instanciamos TriánguloRectángulo la primera vez, pasamos los valores 4 y 5 y en ese caso el `self` es el objeto que terminamos llamando t1 y guarda esos dos valores, y cuando la instanciamos por segunda vez, pasando los valores 10 y 1, `self` es el objeto que terminamos llamando t2, con estos dos otros valores en vez.

Usemos ahora el otro método que tenemos definido:

CELL 10

```
t1.hipotenusa()
```

```
6.4031242374328485
```

Cell 11

```
t2.hipotenusa()
```

```
10.04987562112089
```

Vemos que lo ejecutamos desde t1 y t2, y no pasamos ningún parámetro. Pero en la definición, arriba en la clase, recibe el parámetro `self`. Estamos en la misma situación que antes: como es el método de una clase, cuando lo ejecutamos desde una instancia de la clase Python insertará el objeto mismo como parámetro, que oportunamente usamos para la cuenta: cuando hacemos por ejemplo `self.cateto1 ** 2` estamos usando el nombre `cateto1` de adentro del objeto, que para t1 apuntará a 4 y para t2 apuntará a 10.

Al final de cuentas, lo que tenemos es un determinado código que se aplica a los valores que tiene cada instancia de esa clase. Exactamente el concepto con el que arrancamos arriba toda la explicación de clases, objetos, y POO.

Otro concepto muy útil cuando queremos modelar la realidad usando objetos es el de "herencia", generalmente utilizado cuando tenemos algunos comportamientos que son comunes a distintos tipos de objetos. Lo normal es encontrar una clase "padre" y muchas clases "hijas", pero Python soporta herencia múltiple, aunque es de uso más raro.

Cuando definimos una clase, entonces, si queremos que herede de otra incluiremos a esta última entre paréntesis en la definición de la primera. No vamos a entrar en detalle en este libro sobre cómo explotar todas las características del concepto de herencia, pero lo mencionamos para poder reconocer su uso cuando lo encontremos en algún código.

Particularmente, un caso que incluso ya mostramos en la Sección **??** es cuando creamos una excepción propia, que para que sea justamente una excepción utilizable por el lenguaje, la definimos heredando su comportamiento de alguna excepción integrada en el lenguaje:

Cell 12

```
class NoNegativos(ValueError):
    """Usada al encontrar números negativos, que no se puede."""
```

Para profundizar más sobre la utilización de clases pueden seguir leyendo sobre este tema en el capítulo específico de clases **??**.

## 1.3. Módulos

Como vimos hasta ahora, las formas más comunes de encapsular código son las funciones y las clases. Entonces, pondremos código adentro de esas estructuras, que usaremos desde distintos puntos de nuestro programa. El paso natural siguiente es el de agrupar algunas de esas funciones y clases de nuestro programa en "módulos", de manera de poder importar esos módulos de distintos lugares y tener acceso a las funciones y clases (y cualquier otra estructura) que pongamos allí.

Los módulos no son más que archivos de Python, sin tener casi ninguna otra restricción. Mostremos un ejemplo para ver lo sencillo que es crear y usar un nuevo módulo de Python.

Pongamos el siguiente código en un archivo, al que llamaremos `perimetros.py`:

```
import math

dos_pi = 2 * math.pi

def cuadrado(lado):
    """Calcula el perímetro del cuadrado."""
    return 4 * lado

def círculo(radio):
    """Calcula el perímetro del círculo."""
    return dos_pi * radio
```

Vemos en el ejemplo que además de la definición de esas funciones tenemos otras líneas de código a "nivel de módulo", como el `import` o el cálculo para tener "dos $\pi$" a mano. Todo el código del módulo se ejecutará cuando lo importemos; se importará math, se definirá `dos_pi`, y también se definirán las dos funciones que usaremos luego.

Para usar ese módulo, como es el caso con cualquier otro módulo, sólo tenemos que importarlo. Entonces, en el mismo directorio que grabamos `perimetros.py`, abramos un intérprete interactivo y hagamos:

```
>>> import perimetros
>>> perimetros.círculo(12)
75.39822368615503
```

Necesitamos que el módulo esté en el mismo directorio donde abrimos el intérprete interactivo porque cuando hacemos el `import` Python va a buscar el nombre que indicamos en una serie de directorios, entre ellos el actual del proceso. Claro, podríamos poner nuestro módulo en algunos de los otros directorios donde Python busca, pero ello ya implicaría "instalar" nuestro módulo.

La otra opción para facilitar que podamos encontrar nuestro módulo es directamente agregar el directorio que necesitemos en la lista de lugares donde Python busca. Esto lo podemos hacer a través de la variable de entorno `PYTHONPATH` del sistema operativo, o incluso desde adentro de Python modificando `sys.path`.

Volviendo al ejemplo donde usamos nuestro módulo, vemos que lo importamos usando su nombre, y luego podemos acceder a su contenido usando el `.`, como ya vimos en otros casos. Esta no es la única manera de importar el módulo, también podemos utilizar otra forma en la que en vez de quedarnos con el nombre del módulo para trabajar, nos traemos directamente los nombres de las estructuras internas:

```
>>> from perimetros import círculo
>>> círculo(12)
75.39822368615503
```

Tengamos en cuenta que no cambia nada a la hora de importar el módulo en sí, no es más rápido, ni usa menos memoria, ni ejecuta menos código: la única diferencia es con qué nombres nos quedamos para trabajar.

Bien, ya sabemos agrupar nuestro código en módulos. El próximo paso es agrupar esos módulos en un próximo nivel.

La estructura para agrupar módulos se llama "paquete", que no son más que directorios.

Para probar esto, creemos un directorio `geom` y pongamos nuestro archivo `perimetros.py` allí. Eso es todo; ahora para importar ese módulo tenemos que especificar el paquete. Veamos las distintas formas de terminar ejecutando la función `círculo` en esta nueva situación: indicando el paquete y el módulo, trayendo el módulo del paquete, y trayendo directamente la función (de nuevo, en los tres casos el módulo se importa exactamente igual, sólo cambia con qué nos quedamos para trabajar).

```
>>> import geom.perimetros
>>> geom.perimetros.círculo(12)
75.39822368615503
>>> from geom import perimetros
>>> perimetros.círculo(12)
75.39822368615503
>>> from geom.perimetros import círculo
>>> círculo(12)
75.39822368615503
```

Si en un directorio/paquete ponemos un archivo con el nombre especial `__init__.py`, este archivo se ejecutará cuando importemos el paquete o cualquier módulo de ese paquete, lo cual es muy práctico para cualquier tipo de inicialización que necesitemos realizar.

# 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 [4].

- 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://docs.python.org/es/dev/reference/compound_stmts.html#function-definitions.

[3] URL: https://docs.python.org/dev/reference/datamodel.html#special-method-names.

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



VERSIÓN PRELIMINAR