Contenidos a Trabajar

1. Clases
2. Herencia
3. Excepciones

Clases

Comparado con otros lenguajes de programación, el mecanismo de clases de Python agrega clases con un mínimo de nuevas sintaxis y semánticas. Es una mezcla de los mecanismos de clases encontrados en C++ y Modula-3. Las clases de Python proveen todas las características normales de la Programación Orientada a Objetos: el mecanismo de la herencia de clases permite múltiples clases base, una clase derivada puede sobrescribir cualquier método de su(s) clase(s) base, y un método puede llamar al método de la clase base con el mismo nombre. Los objetos pueden tener una cantidad arbitraria de datos de cualquier tipo. Igual que con los módulos, las clases participan de la naturaleza dinámica de Python: se crean en tiempo de ejecución, y pueden modificarse luego de la creación.

En terminología de C++, normalmente los miembros de las clases (incluyendo los miembros de datos), son *públicos* (excepto ver luego *Variables privadas*), y todas las funciones miembro son *virtuales*. Como en Modula-3, no hay atajos para hacer referencia a los miembros del objeto desde sus métodos: la función método se declara con un primer argumento explícito que representa al objeto, el cual se provee implícitamente por la llamada. Como en Smalltalk, las clases mismas son objetos. Esto provee una semántica para importar y renombrar. A diferencia de C++ y Modula-3, los tipos de datos integrados pueden usarse como clases base para que el usuario los extienda. También, como en C++ pero a diferencia de Modula-3, la mayoría de los operadores integrados con sintaxis especial (operadores aritméticos, de subíndice, etc.) pueden ser redefinidos por instancias de la clase.

Unas palabras sobre nombres y objetos

Los objetos tienen individualidad, y múltiples nombres (en muchos ámbitos) pueden vincularse al mismo objeto. Esto se conoce como *aliasing* en otros lenguajes. Normalmente no se aprecia esto a primera vista en Python, y puede ignorarse sin problemas cuando se maneja tipos básicos inmutables (números, cadenas, tuplas). Sin embargo, el *aliasing*, o renombrado, tiene un efecto posiblemente sorpresivo sobre la semántica de código Python que involucra objetos mutables como listas, diccionarios, y la mayoría de otros tipos. Esto se usa normalmente para beneficio del programa, ya que los renombres funcionan

como punteros en algunos aspectos. Por ejemplo, pasar un objeto es barato ya que la implementación solamente pasa el puntero; y si una función modifica el objeto que fue pasado, el que la llama verá el cambio; esto elimina la necesidad de tener dos formas diferentes de pasar argumentos, como en Pascal u otros lenguajes más antiguos.

Ámbitos y espacios de nombres en Python

Antes de ver clases, primero debo decirte algo acerca de las reglas de ámbito de Python. Las definiciones de clases hacen unos lindos trucos con los espacios de nombres, y necesitás saber cómo funcionan los alcances y espacios de nombres para entender por completo cómo es la cosa. De paso, los conocimientos en este tema son útiles para cualquier programador Python avanzado.

Comencemos con unas definiciones.

Un *espacio de nombres* es una relación de nombres a objetos. Muchos espacios de nombres están implementados en este momento como diccionarios de Python, pero eso no se nota para nada (excepto por el desempeño), y puede cambiar en el futuro. Como ejemplos de espacios de nombres tenés: el conjunto de nombres incluidos (conteniendo funciones como **abs()**, y los nombres de excepciones integradas); los nombres globales en un módulo; y los nombres locales en la invocación a una función. Lo que es importante saber de los espacios de nombres es que no hay relación en absoluto entre los nombres de espacios de nombres distintos; por ejemplo, dos módulos diferentes pueden tener definidos los dos una función maximizar sin confusión; los usuarios de los módulos deben usar el nombre del módulo como prefijo.

Por cierto, yo uso la palabra *atributo* para cualquier cosa después de un punto; por ejemplo, en la expresión z.real, real es un atributo del objeto z. Estrictamente hablando, las referencias a nombres en módulos son referencias a atributos: en la expresión modulo.funcion, modulo es un objeto módulo y funcion es un atributo de éste. En este caso hay una relación directa entre los atributos del módulo y los nombres globales definidos en el módulo: ¡están compartiendo el mismo espacio de nombres!

Los atributos pueden ser de sólo lectura, o de escritura. En el último caso es posible la asignación a atributos. Los atributos de módulo pueden escribirse: modulo.la_respuesta = 42. Los atributos de escritura se pueden borrar también

con la declaración **del**. Por ejemplo, del modulo.la_respuesta va a eliminar el atributo **la_respuesta** del objeto con nombre modulo.

Los espacios de nombres se crean en diferentes momentos y con diferentes tiempos de vida. El espacio de nombres que contiene los nombres incluidos se crea cuando se inicia el intérprete, y nunca se borra. El espacio de nombres global de un módulo se crea cuando se lee la definición de un módulo; normalmente, los espacios de nombres de módulos también duran hasta que el intérprete finaliza. Las instrucciones ejecutadas en el nivel de llamadas superior del intérprete, ya sea desde un script o interactivamente, se consideran parte del módulo llamado **__main__**, por lo tanto tienen su propio espacio de nombres global. (Los nombres incluidos en realidad también viven en un módulo; este se llama **builtins**.)

El espacio de nombres local a una función se crea cuando la función es llamada, y se elimina cuando la función retorna o lanza una excepción que no se maneje dentro de la función. (Podríamos decir que lo que pasa en realidad es que ese espacio de nombres se "olvida".) Por supuesto, las llamadas recursivas tienen cada una su propio espacio de nombres local.

Un *ámbito* es una región textual de un programa en Python donde un espacio de nombres es accesible directamente.

"Accesible directamente" significa que una referencia sin calificar a un nombre intenta encontrar dicho nombre dentro del espacio de nombres.

Aunque los alcances se determinan estáticamente, se usan dinámicamente. En cualquier momento durante la ejecución hay por lo menos cuatro alcances anidados cuyos espacios de nombres son directamente accesibles:

- el ámbito interno, donde se busca primero, contiene los nombres locales
- los espacios de nombres de las funciones anexas, en las cuales se busca empezando por el ámbito adjunto más cercano, contiene los nombres no locales pero también los no globales
- el ámbito anteúltimo contiene los nombres globales del módulo actual
- el ámbito exterior (donde se busca al final) es el espacio de nombres que contiene los nombres incluidos

Si un nombre se declara como global, entonces todas las referencias y asignaciones al mismo van directo al ámbito intermedio que contiene los nombres globales del módulo. Para reasignar nombres encontrados afuera del

ámbito más interno, se puede usar la declaración **nonlocal**; si no se declara nonlocal, esas variables serán de sólo lectura (un intento de escribir a esas variables simplemente crea una *nueva* variable local en el ámbito interno, dejando intacta la variable externa del mismo nombre).

Habitualmente, el ámbito local referencia los nombres locales de la función actual. Fuera de una función, el ámbito local referencia al mismo espacio de nombres que el ámbito global: el espacio de nombres del módulo. Las definiciones de clases crean un espacio de nombres más en el ámbito local.

Es importante notar que los alcances se determinan textualmente: el ámbito global de una función definida en un módulo es el espacio de nombres de ese módulo, no importa desde dónde o con qué alias se llame a la función. Por otro lado, la búsqueda de nombres se hace dinámicamente, en tiempo de ejecución; sin embargo, la definición del lenguaje está evolucionando a hacer resolución de nombres estáticamente, en tiempo de "compilación", ¡así que no te confíes de la resolución de nombres dinámica! (De hecho, las variables locales ya se determinan estáticamente.).

Una peculiaridad especial de Python es que, si no hay una declaración **global** o **nonlocal** en efecto, las asignaciones a nombres siempre van al ámbito interno. Las asignaciones no copian datos, solamente asocian nombres a objetos. Lo mismo cuando se borra: la declaración del x quita la asociación de x del espacio de nombres referenciado por el ámbito local.

De hecho, todas las operaciones que introducen nuevos nombres usan el ámbito local: en particular, las instrucciones **import** y las definiciones de funciones asocian el módulo o nombre de la función al espacio de nombres en el ámbito local.

La declaración **global** puede usarse para indicar que ciertas variables viven en el ámbito global y deberían reasignarse allí; la declaración **nonlocal** indica que ciertas variables viven en un ámbito encerrado y deberían reasignarse allí.

Ejemplo de ámbitos y espacios de nombre

Este es un ejemplo que muestra cómo hacer referencia a distintos ámbitos y espacios de nombres, y cómo las declaraciones **global** y **nonlocal** afectan la asignación de variables:

```
def prueba_ambitos():
    def hacer_local():
        algo = "algo local"
    def hacer_nonlocal():
        nonlocal algo
        algo = "algo no local"
    def hacer_global():
        global algo
        algo = "algo global"
    algo = "algo de prueba"
    hacer_local()
    print("Luego de la asignación local:", algo)
    hacer_nonlocal()
    print("Luego de la asignación no local:", algo)
    hacer_global()
    print("Luego de la asignación global:", algo)

prueba_ambitos()
print("In global scope:", algo)
```

El resultado del código ejemplo es:

```
Luego de la asignación local: algo de prueba
Luego de la asignación no local: algo no local
Luego de la asignación global: algo no local
En el ámbito global: algo global
```

Notar como la asignación *local* (que es el comportamiento normal) no cambió la vinculación de *algo* de *prueba_ambitos*. La asignación **nonlocal** cambió la vinculación de *algo* de *prueba_ambitos*, y la asignación **global** cambió la vinculación a nivel de módulo.

También podes ver que no había vinculación para *algo* antes de la asignación **global**.

Sintaxis de definición de clases

La forma más sencilla de definición de una clase se ve así:

```
class Clase:
    <declaración-1>
    .
    .
    .
    <declaración-N>
```

Las definiciones de clases, al igual que las definiciones de funciones (instrucciones **def**) deben ejecutarse antes de que tengan efecto alguno. (Es concebible poner una definición de clase dentro de una rama de un **if**, o dentro de una función.)

En la práctica, las declaraciones dentro de una clase son definiciones de funciones, pero otras declaraciones son permitidas, y a veces resultan útiles; veremos esto más adelante. Las definiciones de funciones dentro de una clase normalmente tienen una lista de argumentos peculiar, dictada por las convenciones de invocación de métodos; a esto también lo veremos más adelante.

Cuando se ingresa una definición de clase, se crea un nuevo espacio de nombres, el cual se usa como ámbito local; por lo tanto, todas las asignaciones a variables locales van a este nuevo espacio de nombres. En particular, las definiciones de funciones asocian el nombre de las funciones nuevas allí.

Cuando una definición de clase se finaliza normalmente se crea un *objeto clase*. Básicamente, este objeto envuelve los contenidos del espacio de nombres creado por la definición de la clase; aprenderemos más acerca de los objetos clase en la sección siguiente. El ámbito local original (el que tenía efecto justo antes de que ingrese la definición de la clase) es restablecido, y el objeto clase se asocia allí al nombre que se le puso a la clase en el encabezado de su definición (**Clase** en el ejemplo)

Objetos clase

Los objetos clase soportan dos tipos de operaciones: hacer referencia a atributos e instanciación.

Para *hacer referencia a atributos* se usa la sintaxis estándar de todas las referencias a atributos en Python:

objeto.nombre. Los nombres de atributo válidos son todos los nombres que estaban en el espacio de nombres de la clase cuando ésta se creó. Por lo tanto, si la definición de la clase es así:

```
class MiClase:
    """Simple clase de ejemplo"""
    i = 12345
    def f(self):
        return 'hola mundo'
```

...entonces MiClase.i y MiClase.f son referencias de atributos válidas, que devuelven un entero y un objeto función respectivamente. Los atributos de clase también pueden ser asignados, o sea que podés cambiar el valor de MiClase.i mediante asignación. **__doc__** también es un atributo válido, que devuelve la documentación asociada a la clase:

"Simple clase de ejemplo".

La *instanciación* de clases usa la notación de funciones. Hacé de cuenta que el objeto de clase es una función sin parámetros que devuelve una nueva instancia de la clase. Por ejemplo (para la clase de más arriba):

```
x = MiClase()
```

...crea una nueva *instancia* de la clase y asigna este objeto a la variable local x. La operación de instanciación ("llamar" a un objeto clase) crea un objeto vacío. Muchas clases necesitan crear objetos con instancias en un estado inicial particular. Por lo tanto una clase puede definir un método especial llamado **__init__()**, de esta forma:

```
def __init__(self):
    self.datos = []
```

Cuando una clase define un método **__init__()**, la instanciación de la clase automáticamente invoca a **__init__()** para la instancia recién creada. Entonces, en este ejemplo, una instancia nueva e inicializada se puede obtener haciendo:

```
x = MiClase()
```

Por supuesto, el método **__init__()** puede tener argumentos para mayor flexibilidad. En ese caso, los argumentos que se pasen al operador de instanciación de la clase van a parar al método **__init__()**. Por ejemplo,

```
>>> class Complejo:
...     def __init__(self, partereal, parteimaginaria):
...         self.r = partereal
...         self.i = parteimaginaria
...
>>> x = Complejo(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Objetos método

Generalmente, un método es llamado luego de ser vinculado:

```
x . f()
```

En el ejemplo **MiClase**, esto devuelve la cadena 'hola mundo'. Pero no es necesario llamar al método justo en ese momento: x.f es un objeto método, y puede ser guardado y llamado más tarde. Por ejemplo:

```
xf = x.f
while True:
    print(xf())
```

...continuará imprimiendo hola mundo hasta el fin de los días.

¿Qué sucede exactamente cuando un método es llamado? Debés haber notado que x.f() fue llamado más arriba sin ningún argumento, a pesar de que la definición de función de **f()** especificaba un argumento. ¿Qué pasó con ese argumento? Seguramente Python levanta una excepción cuando una función que requiere un argumento es llamada sin ninguno, aún si el argumento no es utilizado...

De hecho, tal vez hayas adivinado la respuesta: lo que tienen de especial los métodos es que el objeto es pasado como el primer argumento de la función. En nuestro ejemplo, la llamada x.f() es exactamente equivalente a MiClase.f(x). En general, llamar a un método con una lista de *n* argumentos es equivalente a

llamar a la función correspondiente con una lista de argumentos que es creada insertando el objeto del método antes del primer argumento.

Si aún no comprendés cómo funcionan los métodos, un vistazo a la implementación puede ayudar a clarificar este tema.

Cuando se hace referencia un atributo de instancia y no es un atributo de datos, se busca dentro de su clase. Si el nombre denota un atributo de clase válido que es un objeto función, se crea un objeto método juntando (punteros a) el objeto instancia y el objeto función que ha sido encontrado. Este objeto abstracto creado de esta unión es el objeto método.

Cuando el objeto método es llamado con una lista de argumentos, una lista de argumentos nueva es construida a partir del objeto instancia y la lista de argumentos original, y el objeto función es llamado con esta nueva lista de argumentos

Variables de clase y de instancia

En general, las variables de instancia son para datos únicos de cada instancia y las variables de clase son para atributos y métodos compartidos por todas las instancias de la clase:

```
class Perro:

    tipo = 'canino'                 # variable de clase compartida por todas las instancias

    def __init__(self, nombre):
        self.nombre = nombre        # variable de instancia única para la instancia

>>> d = Perro('Fido')
>>> e = Perro('Buddy')
>>> d.tipo                          # compartido por todos los perros
'canino'
>>> e.tipo                          # compartido por todos los perros
'canino'
>>> d.nombre                        # único para d
'Fido'
>>> e.nombre                        # único para e
'Buddy'
```

Como se vió en ***Unas palabras sobre nombres y objetos***, los datos compartidos pueden tener efectos inesperados que involucren objetos *mutables* como ser listas y diccionarios. Por ejemplo, la lista *trucos* en el siguiente código no debería

ser usada como variable de clase porque una sola lista sería compartida por todos las instancias de *Perro*:

```
class Perro:

    trucos = []         # uso incorrecto de una variable de clase

    def __init__(self, nombre):
        self.nombre = nombre

    def agregar_truco(self, truco):
        self.trucos.append(truco)

>>> d = Perro('Fido')
>>> e = Perro('Buddy')
>>> d.agregar_truco('girar')
>>> e.agregar_truco('hacerse el muerto')
>>> d.trucos                # compartidos por todos los perros inesperadamente
['girar', 'hacerse el muerto']
```

El diseño correcto de esta clase sería usando una variable de instancia:

```
class Perro:
def __init__(self, nombre):
        self.nombre = nombre
        self.trucos = []# crea una nueva lista vacía para cada perro
def agregar_truco(self, truco):
        self.trucos.append(truco)
>>> d = Perro('Fido')
>>> e = Perro('Buddy')
>>> d.agregar_truco('girar')
>>> e.agregar_truco('hacerse el muerto')
>>> d.trucos
['girar']
>>> e.trucos
['hacerse el muerto']
```

Herencia

Por supuesto, una característica del lenguaje no sería digna del nombre "clase" si no soportara herencia. La sintaxis para una definición de clase derivada se ve así:

```
class ClaseDerivada(ClaseBase):
    <declaración-1>
    .
    .
    .
    <declaración-N>
```

El nombre **ClaseBase** debe estar definido en un ámbito que contenga a la definición de la clase derivada. En el lugar del nombre de la clase base se permiten otras expresiones arbitrarias. Esto puede ser útil, por ejemplo, cuando la clase base está definida en otro módulo:

```
class ClaseDerivada(modulo.ClaseBase):
```

La ejecución de una definición de clase derivada procede de la misma forma que una clase base. Cuando el objeto clase se construye, se tiene en cuenta a la clase base. Esto se usa para resolver referencias a atributos: si un atributo solicitado no se encuentra en la clase, la búsqueda continua por la clase base. Esta regla se aplica recursivamente si la clase base misma deriva de alguna otra clase.

No hay nada en especial en la instanciación de clases derivadas: ClaseDerivada() crea una nueva instancia de la clase.

Las referencias a métodos se resuelven de la siguiente manera: se busca el atributo de clase correspondiente, descendiendo por la cadena de clases base si es necesario, y la referencia al método es válida si se entrega un objeto función.

Las clases derivadas pueden redefinir métodos de su clase base. Como los métodos no tienen privilegios especiales cuando llaman a otros métodos del mismo objeto, un método de la clase base que llame a otro método definido en la misma clase base puede terminar llamando a un método de la clase derivada que lo haya redefinido. (Para los programadores de C++: en Python todos los métodos son en efecto virtuales.)

Un método redefinido en una clase derivada puede de hecho querer extender en vez de simplemente reemplazar al método de la clase base con el mismo nombre. Hay una manera simple de llamar al método de la clase base directamente:

simplemente llamás a ClaseBase.metodo(self, argumentos). En ocasiones esto es útil para los clientes también. (Observá que esto sólo funciona si la clase base es accesible como CalseBase en el ámbito global.)

Python tiene dos funciones integradas que funcionan con herencia:

- Usar **isinstance()** para verificar el tipo de una instancia: isinstance(obj, int) devuelve True solo si obj.__class__ es **int** o alguna clase derivada de **int**.
- Usar **issubclass()** para comprobar herencia de clase: issubclass(bool, int) da True ya que **bool** es una subclase de **int**. Sin embargo, issubclass(float, int) devuelve False porque **float** no es una subclase de **int**.

Herencia múltiple

Python también soporta una forma de herencia múltiple. Una definición de clase con múltiples clases base se ve así:

```
class ClaseDerivada(Base1, Base2, Base3):
    <declaración-1>
    .
    .
    .
    <declaración-N>
```

Para la mayoría de los propósitos, en los casos más simples, podés pensar en la búsqueda de los atributos heredados de clases padres como primero en profundidad, de izquierda a derecha, sin repetir la misma clase cuando está dos veces en la jerarquía. Por lo tanto, si un atributo no se encuentra en **ClaseDerivada**, se busca en **Base1**, luego (recursivamente) en las clases base de **Base1**, y sólo si no se encuentra allí se lo busca en **Base2**, y así sucesivamente.

En realidad es un poco más complejo que eso; el orden de resolución de métodos cambia dinámicamente para soportar las llamadas cooperativas a **super()**. Este enfoque es conocido en otros lenguajes con herencia múltiple como "llámese al siguiente método" y es más poderoso que la llamada al superior que se encuentra en lenguajes con sólo herencia simple.

El ordenamiento dinámico es necesario porque todos los casos de herencia múltiple exhiben una o más relaciones en diamante (cuando se puede llegar al

menos a una de las clases base por distintos caminos desde la clase de más abajo).

Por ejemplo, todas las clases heredan de **object**, por lo tanto cualquier caso de herencia múltiple provee más de un camino para llegar a **object**. Para que las clases base no sean accedidas más de una vez, el algoritmo dinámico hace lineal el orden de búsqueda de manera que se preserve el orden de izquierda a derecha especificado en cada clase, que se llame a cada clase base sólo una vez, y que sea monótona (lo cual significa que una clase puede tener clases derivadas sin afectar el orden de precedencia de sus clases bases). En conjunto, estas propiedades hacen posible diseñar clases confiables y extensibles con herencia múltiple.

Variables privadas

Las variables "privadas" de instancia, que no pueden accederse excepto desde dentro de un objeto, no existen en Python.

Sin embargo, hay una convención que se sigue en la mayoría del código Python: un nombre prefijado con un guión bajo (por ejemplo, _spam) debería tratarse como una parte no pública de la API (más allá de que sea una función, un método, o un dato). Debería considerarse un detalle de implementación y que está sujeto a cambios sin aviso.

Ya que hay un caso de uso válido para los identificadores privados de clase (a saber: colisión de nombres con nombres definidos en las subclases), hay un soporte limitado para este mecanismo. Cualquier identificador con la forma __spam (al menos dos guiones bajos al principio, como mucho un guión bajo al final) es textualmente reemplazado por _nombredeclase__spam, donde nombredeclase es el nombre de clase actual al que se le sacan guiones bajos del comienzo (si los tuviera). Se modifica el nombre del identificador sin importar su posición sintáctica, siempre y cuando ocurra dentro de la definición de una clase.

La modificación de nombres es útil para dejar que las subclases sobreescriban los métodos sin romper las llamadas a los métodos desde la misma clase. Por ejemplo:

```python
class Mapeo:
    def __init__(self, iterable):
        self.lista_de_items = []
        self.__actualizar(iterable)

    def actualizar(self, iterable):
        for item in iterable:
            self.lista_de_items.append(item)

    __actualizar = actualizar   # copia privada del actualizar() original

class SubClaseMapeo(Mapeo):

    def actualizar(self, keys, values):
        # provee una nueva signatura para actualizar()
        # pero no rompe __init__()
        for item in zip(keys, values):
            self.lista_de_items.append(item)
```

Hay que aclarar que las reglas de modificación de nombres están diseñadas principalmente para evitar accidentes; es posible acceder o modificar una variable que es considerada como privada. Esto hasta puede resultar útil en circunstancias especiales, tales como en el depurador.

Notar que el código pasado a exec o eval() no considera que el nombre de clase de la clase que invoca sea la clase actual; esto es similar al efecto de la sentencia global, efecto que es de similar manera restringido a código que es compilado en conjunto. La misma restricción aplica a getattr(), setattr() y delattr(), así como cuando se referencia a __dict__ directamente.

Excepciones

Una **excepción** es el bloque de código que se lanza cuando se produce un **error** en la ejecución de un programa Python.

De hecho ya hemos visto algunas de estas excepciones: accesos fuera de rango a listas o tuplas, accesos a claves inexistentes en diccionarios, etc. Cuando ejecutamos código que podría fallar bajo ciertas circunstancias, necesitamos también manejar, de manera adecuada, las excepciones que se generan.

Manejando errores

Si una excepción ocurre en algún lugar de nuestro programa y no es capturada en ese punto, va subiendo (burbujeando) hasta que es capturada en alguna función que ha hecho la llamada. Si en toda la «pila» de llamadas no existe un control de la excepción, Python muestra un mensaje de error con información adicional:

```
>>> def intdiv(a, b):
...     return a // b
...

>>> intdiv(3, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in intdiv
ZeroDivisionError: integer division or modulo by zero
```

Para manejar (capturar) las excepciones podemos usar un bloque de código con las palabras reservadas try and except:

```
>>> def intdiv(a, b):
...     try:
...         return a // b
...     except:
...         print('Please do not divide by zero...')
...

>>> intdiv(3, 0)
Please do not divide by zero...
```

Agencia de Aprendizaje a lo largo de la vida

Aquel código que se encuentre dentro del bloque try se ejecutará normalmente siempre y cuando no haya un error. Si se produce una excepción, ésta será capturada por el bloque except, ejecutándose el código que contiene.

Consejo: No es una buena práctica usar un bloque except sin indicar el **tipo de excepción** que estamos gestionando, no sólo porque puedan existir varias excepciones que capturar sino porque, como dice el "Zen de Python": «explícito» es mejor que «implícito». Para visitar mas sobre el zen de Python, vaya al sitio: https://peps.python.org/pep-0020/

Especificando excepciones

En el siguiente ejemplo mejoraremos el código anterior, capturando distintos tipos de excepciones:

- TypeError por si los operandos no permiten la división.
- ZeroDivisionError por si el denominador es cero.
- Exception para cualquier otro error que se pueda producir.

Veamos su implementación:

```
>>> def intdiv(a, b):
...     try:
...         result = a // b
...     except TypeError:
...         print('Check operands. Some of them seems strange...')
...     except ZeroDivisionError:
...         print('Please do not divide by zero...')
...     except Exception:
...         print('Ups. Something went wrong...')
...

>>> intdiv(3, 0)
Please do not divide by zero...

>>> intdiv(3, '0')
Check operands. Some of them seems strange...
```

Cubriendo más casos

Python proporciona la cláusula else para saber que todo ha ido bien y que no se ha lanzado ninguna excepción. Esto es relevante a la hora de manejar los errores.

De igual modo, tenemos a nuestra disposición la cláusula finally que se ejecuta siempre, independientemente de si ha habido o no ha habido error.

Veamos un ejemplo de ambos:

```
>>> values = [4, 2, 7]

>>> user_index = 3

>>> try:
...     r = values[user_index]
... except IndexError:
...     print('Error: Index not in list')
... else:
...     print(f'Your wishes are my command: {r}')
... finally:
...     print('Have a good day!')
...
Error: Index not in list
Have a good day!

>>> user_index = 2

>>> try:
...     r = values[user_index]
... except IndexError:
...     print('Error: Index not in list')
... else:
...     print(f'Your wishes are my command: {r}')
... finally:
...     print('Have a good day!')
...
Your wishes are my command: 7
Have a good day!
```

Excepciones propias

Python ofrece una gran cantidad de **excepciones predefinidas**. Hasta ahora hemos visto cómo gestionar y manejar este tipo de excepciones. Pero hay ocasiones en las que nos puede interesar crear nuestras propias excepciones. Para ello tendremos que crear una clase **heredando** de **Exception**, la clase base para todas las excepciones.

Veamos un ejemplo en el que creamos una excepción propia controlando que el valor sea un número entero:

```
>>> class NotIntError(Exception):
...     pass
...

>>> values = (4, 7, 2.11, 9)

>>> for value in values:
...     if not isinstance(value, int):
...         raise NotIntError(value)
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
__main__.NotIntError: 2.11
```

Hemos usado la sentencia raise para «elevar» esta excepción, que podría ser controlada en un nivel superior mediante un bloque try – except

Mensaje personalizado

Podemos personalizar la excepción añadiendo un mensaje más informativo. Siguiendo el ejemplo anterior, veamos cómo introducimos esta información:

```
>>> class NotIntError(Exception):
...     def __init__(self, message='This module only works with integers. Sorry!'):
...             super().__init__(message)
...

>>> raise NotIntError()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.NotIntError: This module only works with integers. Sorry!
```

Podemos ir un paso más allá e incorporar en el mensaje el propio valor que está generando el error:

```
>>> class NotIntError(Exception):
...     def __init__(self, value, message='This module only works with integers. Sorry!'):
...         self.value = value
...         self.message = message
...         super().__init__(self.message)
...
...     def __str__(self):
...         return f'{self.value} -> {self.message}'
...

>>> raise NotIntError(2.11)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.NotIntError: 2.11 -> This module only works with integers. Sorry!
```

Aserciones

Si hablamos de control de errores hay que citar una sentencia en Python denominada assert. Esta sentencia nos permite comprobar si se están cumpliendo las «expectativas» de nuestro programa, y en caso contrario, lanza una excepción informativa.

Su sintaxis es muy simple. Únicamente tendremos que indicar una expresión de comparación después de la sentencia:

```
>>> result = 10

>>> assert result > 0

>>> print(result)
10
```

En el caso de que la condición se cumpla, no sucede nada: el programa continúa con su flujo normal. Esto es indicativo de que las expectativas que teníamos se han satisfecho.

Sin embargo, si la condición que fijamos no se cumpla, la aserción devuelve un error AssertionError y el programa interrumpe su ejecución:

```
>>> result = -1

>>> assert result > 0
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-29-e2efe60b0c46> in <module>
----> 1 assert result > 0

AssertionError:
```

Podemos observar que la excepción que se lanza no contiene ningún mensaje informativo. Es posible personalizar este mensaje añadiendo un segundo elemento en la tupla de la aserción:

```
>>> assert result > 0, 'El resultado debe ser positivo'
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-31-f58052ce672b> in <module>
----> 1 assert result > 0, 'El resultado debe ser positivo'

AssertionError: El resultado debe ser positivo
```