Módulos, Paquetes y Excepciones

Python 2019

Aprenderá sobre:
Módulos de Python:

Su lógica, función, cómo importarlos de diferentes maneras y presentar el contenido de algunos módulos estándar proporcionados por Python;

La forma en que los módulos se acoplan para formar paquetes.

El concepto de una excepción y la implementación de Python, incluida la instrucción try-except, con sus aplicaciones.

Introducción

Un código más grande siempre significa un mantenimiento más difícil.

La búsqueda de errores siempre es más fácil cuando el código es más pequeño (al igual que encontrar una rotura mecánica es más simple cuando la maquinaria es más simple y más pequeña).

Además, cuando se espera que el código que se crea sea realmente grande se espera dividirlo en muchas partes, implementado en paralelo por unas pocos, docenas, varias docenas o incluso varios cientos de desarrolladores individuales.

Podemos decir que el código que no crece probablemente sea completamente inutilizable o abandonado.

Introducción (cont)

Si desea que dicho proyecto de software se complete con éxito, debe tener los medios que le permitan:

- **Dividir todas las tareas entre los desarrolladores;**
- **Unir todas las partes creadas en un todo funcional.**

Por ejemplo, un determinado proyecto se puede dividir en dos partes principales:

1. **La interfaz de usuario (la parte que se comunica con el usuario mediante widgets y una pantalla gráfica).**
2. **La lógica (la parte que procesa datos y produce resultados)**

Cada una de estas partes se puede dividir en otras más pequeñas, y así sucesivamente. Tal proceso a menudo se llama descomposición.

Entonces:

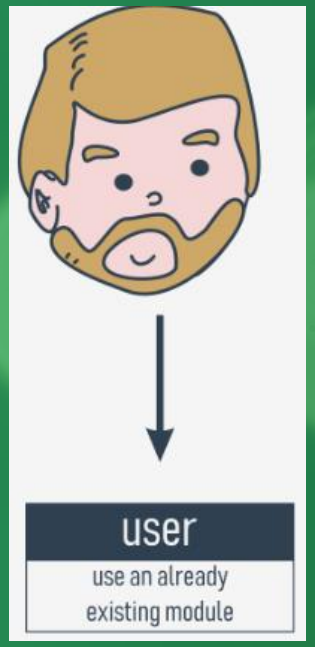
¿Cómo se divide una pieza de software en partes separadas pero cooperantes?

Los módulos son la respuesta.

El concepto Usuario vs. Proveedor

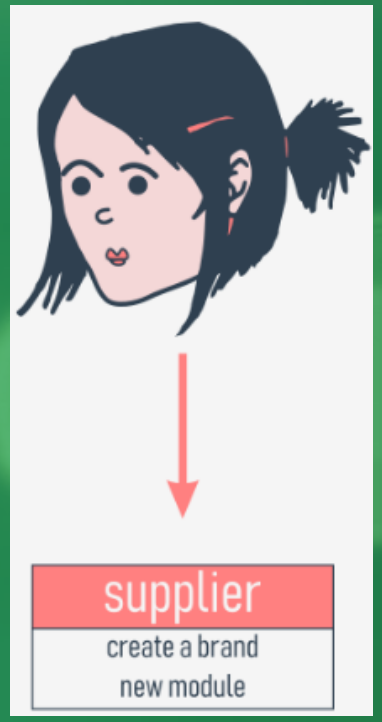
El manejo de los módulos consta de dos cuestiones diferentes:

- **El primero (probablemente el más común) ocurre cuando desea utilizar un módulo ya existente, escrito por otra persona o creado por usted mismo durante su trabajo en algún proyecto complejo; en este caso, usted es el usuario del módulo ;**



El concepto Usuario vs. Proveedor

El segundo ocurre cuando desea crear un módulo nuevo, ya sea para su propio uso o para facilitar la vida de otros programadores: usted es el proveedor del módulo .



¿Cómo hacer uso de un módulo?

En primer lugar, un módulo se identifica por su nombre . Si desea utilizar cualquier módulo, necesita saber el nombre.

Se integra una gran cantidad de módulos junto con Python. Todos estos módulos, junto con las funciones integradas, forman la biblioteca estándar de Python , un tipo especial de biblioteca donde los módulos desempeñan el papel de libros (incluso podemos decir que las carpetas desempeñan el papel de estanterías).

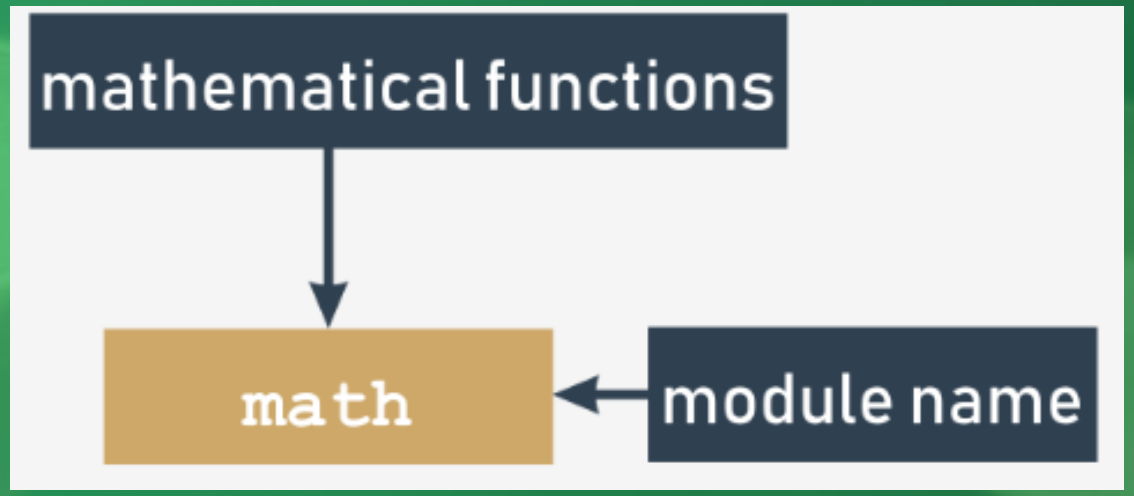
Si desea ver la lista completa de todos los "volúmenes" recopilados en esa biblioteca, puede encontrarla aquí: https://docs.python.org/3/library/index.html.

¿Cómo hacer uso de un módulo?

Cada módulo consta de entidades (como un libro consta de capítulos).

Estas entidades pueden ser funciones, variables, constantes, clases y objetos.

Si sabe cómo acceder a un módulo en particular, puede utilizar cualquiera de las entidades que almacena.



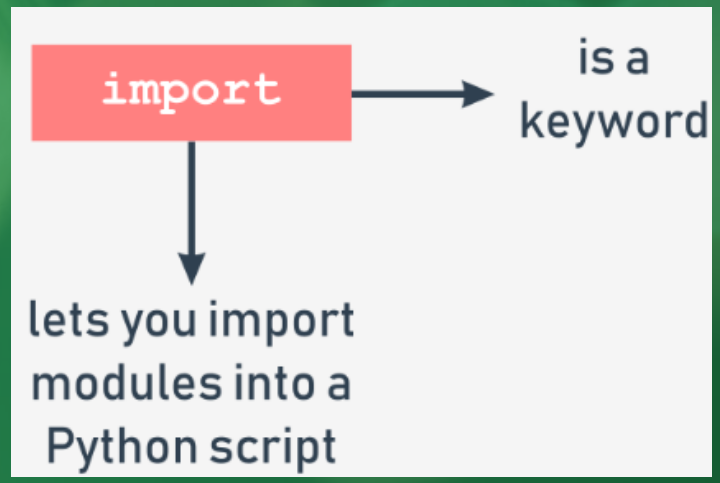
Acceso a módulos: math

Su nombre habla por sí mismo: el módulo contiene una rica colección de entidades (no solo funciones) que permiten a un programador implementar efectivamente cálculos que exigen el uso de funciones matemáticas, como sin () o log () .

Para que un módulo sea utilizable, debe importarlo (piense en ello como sacar un libro del estante).

La importación de un módulo se realiza mediante una instrucción denominada import.

Nota: import también es una palabra clave.



Acceso a módulos: math

Supongamos que desea utilizar dos entidades proporcionadas por el módulo math:

1. **Un símbolo (constante) que representa un valor preciso (tan preciso como sea posible usando aritmética de punto flotante doble) de π (aunque usar una letra griega para nombrar una variable es totalmente posible en Python, el símbolo se llama pi ; es una solución más conveniente , especialmente para esa parte del mundo que ni tiene ni va a usar un teclado griego)**
2. **Una función llamada sin() (el equivalente en computadora de la función seno matemática)**

Ambas entidades están disponibles a través del módulo math, pero la forma en que puede usarlas depende en gran medida de cómo se haya realizado la importación.

La forma más sencilla de importar un módulo en particular es usar la instrucción de importación de la siguiente manera:

import math

import math

La comando debe tener las siguientes condiciones contiene:

1. La palabra clave ***import***
2. El nombre del módulo que es sujeto a importación.

La instrucción puede ubicarse en cualquier parte de su código, pero debe colocarse antes del primer uso de cualquiera de las entidades del módulo .

Si desea (o tiene que) importar más de un módulo, puede hacerlo repitiendo la instrucción import o enumerando los módulos después de la palabra **import x**, como aquí:

import math, sys

La instrucción importa dos módulos, primero el llamado math y luego el segundo nombrado sys.

La lista de módulos puede ser larga.

Namespace ?

Un ***namespace*** es un espacio (entendido en un contexto no físico) en el que existen algunos nombres y los nombres no entran en conflicto entre sí (es decir, no hay dos objetos diferentes con el mismo nombre).

Podemos decir que cada grupo social es un namespace: el grupo tiende a nombrar a cada uno de sus miembros de una manera única (por ejemplo, los padres no darán a sus hijos los mismos nombres).

Dentro de un determinado namespace, cada nombre debe seguir siendo único .

Esto puede significar que algunos nombres pueden desaparecer cuando cualquier otra entidad de un nombre ya conocido ingresa al namespace.

Si el módulo de un nombre especificado existe y es accesible Python importa su contenido, es decir, todos los nombres definidos en el módulo se conocen , pero no ingresan el namespace de su código.

Esto significa que puede tener sus propias entidades nombradas SIN() y PI () y no se verán afectadas por la importación de ninguna manera.

El concepto de namespace: math y pi

En este punto, es posible que se pregunte cómo acceder a lo que viene de la funcionalidad de PI en el módulo math.

Para hacer esto, debe asociar la llamada de pi con el nombre de su módulo original.

Importación de PI en math

Mire el fragmento a continuación, esta es la forma en que califica los nombres de pi y sin con el nombre de su módulo de origen:

math.pi

math.sin

Es simple, se usa:

1. el nombre del módulo (math)
2. un punto .
3. el nombre de la entidad (pi)

Tal forma que indica claramente el namespace en el que existe el nombre de l afuncionalidad que deseamos usar, en este caso PI y SIN en el modulo math.

Importación de PI en math

En el segundo método, la sintaxis de **import** señala con precisión qué entidad (o entidades) del módulo son aceptables en el código:

from math import pi

La instrucción consta de los siguientes elementos:

1. la palabra clave from
2. el nombre del módulo que se importará (selectivamente);
3. la palabra clave import
4. el nombre o la lista de nombres de la entidad / entidades que se importan al namespace.

La instrucción anterior tendría este efecto:

- las entidades enumeradas (y solo aquellas) se importan del módulo indicado ;
- Se puede acceder a los nombres de las entidades importadas sin calificación .

Nota: no se importan otras entidades. Además, no puede importar entidades adicionales utilizando una calificación, una línea como esta:

Reescribamos el script anterior para incorporar la nueva técnica.

Ejemplo:

```
from math import sin, pi


print(sin(pi/2))
```

Importación *

En el tercer método, de la sintaxis de import una forma más agresiva de las presentada anteriormente:

from module import *

Como puede ver, el nombre de una entidad (o la lista de nombres de entidades) se reemplaza con un solo asterisco (*).

Dicha instrucción importa todas las entidades del módulo indicado.

- **¿Es conveniente? Sí, lo es, ya que lo libera del deber de enumerar todos los nombres que necesita.**
- **¿Es inseguro? Sí, a menos que conozca todos los nombres proporcionados por el módulo, es posible que no pueda evitar conflictos de nombres . Trátelo como una solución temporal e intente no usarlo en el código normal.**

Importar un módulo: la palabra clave as

Si usa la variante de módulo de importación y no le gusta el nombre de un módulo en particular (por ejemplo, es el mismo que uno de sus entidades ya definidas, por lo que su uso se vuelve problemática) puede darle el nombre que desee, esto se llama alias .

- El alias hace que el módulo se identifique con un nombre diferente al original. Esto también puede acortar los nombres calificados.
- La creación de un alias se realiza junto con la importación del módulo, y exige la siguiente forma de la instrucción de importación:

import module as alias

El "módulo" identifica el nombre del módulo original, mientras que el "alias" es el nombre que desea utilizar en lugar del original.

Nota: as es una palabra clave.

AS

Si necesita cambiar la palabra math, puede introducir su propio nombre, como en el ejemplo:

```
import math as m
print(m.sin(m.pi/2))
```

Nota: después de la ejecución exitosa de una importación con alias, el nombre del módulo original se vuelve inaccesible y no debe usarse.

A su vez, cuando usa la variante *from module import name* y necesita cambiar el nombre de la entidad, cree un alias para la entidad. Esto hará que el nombre sea reemplazado por el alias que elija.

Así es como se puede hacer:

from module import name as alias

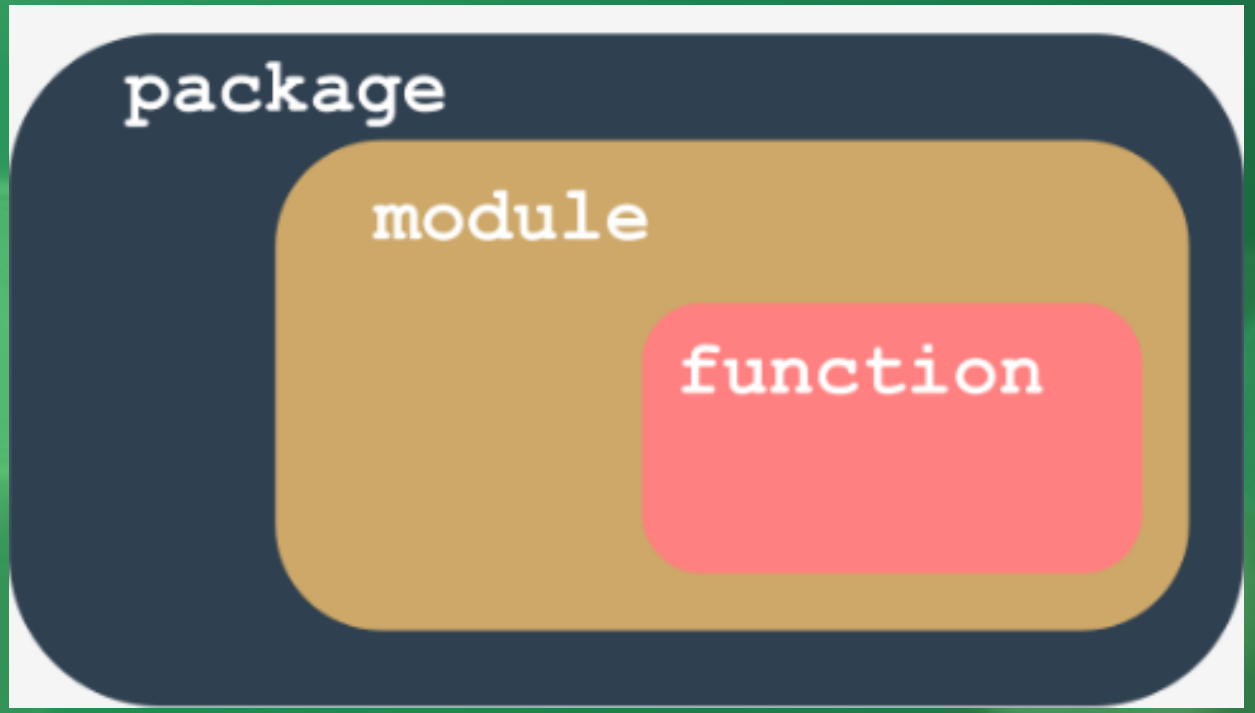
Como anteriormente, el nombre original (sin alias) se vuelve inaccesible.

La frase **name as alias** se puede repetir: use comas para separar las frases multiplicadas. Ejemplo:

```
from module import n as a, m as b, o as c
```

Paquetes en Python

Introducción



¿Qué es un paquete?

Escribir sus propios módulos no difiere mucho de escribir scripts comunes.

Hay algunos aspectos específicos que debe tener en cuenta, pero definitivamente no es ciencia espacial.

algunos asuntos importantes:

- un módulo es un tipo de contenedor lleno de funciones : puede empaquetar tantas funciones como desee en un módulo y distribuirlo por todo el mundo;
- no mezclar funciones con diferentes áreas de aplicación dentro de un modulo, así que agrupe sus funciones cuidadosamente y asigne un nombre al módulo que las contiene.
- hacer muchos módulos puede causar un pequeño desorden: tarde o temprano querrá agrupar sus módulos exactamente de la misma manera que ha agrupado previamente las funciones,

¿Qué es un paquete?

¿habría un contenedor más general que un módulo?

sí, el paquete ; En el mundo de los módulos, un paquete juega un papel similar a una carpeta / directorio, en el mundo de los archivos.

Entonces:

Un paquete es una estructura jerárquica de directorios de archivos que define un único entorno de aplicación.

Ejemplo Python esta compuesto de paquetes, subpaquetes y subpaquetes, y así sucesivamente que construyen todo un sistema

Mi primer modulo

Se necesitan de dos archivos para el ejemplo:

module.py

```
```

main.py

```
import module
```

Para la recreación del ejemplo se necesitan dos:

Uno de ellos será el módulo en sí. Está vacío ahora. No te preocupes, lo llenaremos con el código real.

Hemos llamado al archivo ***module.py*** . No muy creativo, pero simple y claro.

El segundo archivo contiene el código usando el nuevo módulo.

Su nombre es ***main.py***.

Y solo tendrá la instrucción de:

import module

Nota: ambos archivos deben ubicarse en la misma carpeta . Le recomendamos encarecidamente que cree una carpeta nueva y vacía para ambos archivos. Algunas cosas serán más fáciles entonces.

Inicie IDLE y ejecute el archivo main.py.

¿Que se obtuvo como resultado de la ejecución?

No deberías ver nada. Esto significa que Python ha importado con éxito el contenido del archivo module.py .

El primer paso ya está hecho, pero antes de dar el siguiente paso, queremos que eche un vistazo a la carpeta en la que existen ambos archivos.

¿Notas algo interesante?

Ha aparecido una nueva subcarpeta, ¿puede verla? Su nombre es __**pycache**__ . Echa un vistazo al interior. ¿Que ves?

Hay un archivo llamado (más o menos) **module.cpython-xy.pyc** donde **x e y** son dígitos derivados de su versión de Python (por ejemplo, serán **3 y 7 si usa Python 3.7**).

¿Que se obtuvo como resultado de la ejecución?

Puede mirar dentro del archivo: el contenido es completamente ilegible para los humanos. Tiene que ser así, ya que el archivo está destinado solo para uso de Python.

Cuando Python importa un módulo por primera vez, traduce su contenido en una forma algo compilada .

El archivo no contiene código de máquina: es un código interno semi-compilado de Python , listo para ser ejecutado por el intérprete de Python. Como tal archivo no requiere muchas de las comprobaciones necesarias para un archivo fuente puro, la ejecución comienza más rápido y también se ejecuta más rápido.

Gracias a eso, cada importación posterior será más rápida que interpretar el texto fuente desde cero.

Python puede verificar si el archivo fuente del módulo ha sido modificado (en este caso, el archivo pyc se reconstruirá) o no (cuando el archivo pyc se puede ejecutar de una vez). Como este proceso es completamente automático y transparente, no tiene que tenerlo en cuenta.

ejemplo

Ahora modifiquemos el archivo modulo.py

Ponga una línea que haga algo simple:

print ("Quiero ser un modulo")

Que ocurre al ejecutar main.py?

Cuando se importa un módulo, Python ejecuta implícitamente su contenido.

Le da al módulo la oportunidad de inicializar algunos de sus aspectos internos (por ejemplo, puede asignar algunas variables con valores útiles).

Nota: la inicialización se realiza solo una vez , cuando se produce la primera importación, por lo que las asignaciones realizadas por el módulo no se repiten innecesariamente.

Imagine el siguiente contexto:

hay un módulo llamado mod1 ;

hay un módulo llamado mod2 que contiene la instrucción import mod1;

hay un archivo principal que contiene instrucciones import mod1e import mod2.

A primera vista, puede pensar que mod1 se importará dos veces; afortunadamente, solo se produce la primera importación . Python recuerda los módulos importados y silenciosamente omite todas las importaciones posteriores.

Ejemplo

Modifique el archivo modulo.py con el siguiente código:

```
if __name__ == "__main__":
        print("I prefer to be a module")
else:
        print("I like to be a module")
```

Ejecute ahora modulo.py y luego main.py

Puede predecir el resultado??

Crear Paquetes

En Python, cada uno de nuestros archivos .py se denominan módulos.

Estos módulos, a la vez, pueden formar parte de paquetes.

Un paquete, es una carpeta que contiene archivos .py.

Pero, para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado __init__.py.

Este archivo, no necesita contener ninguna instrucción.

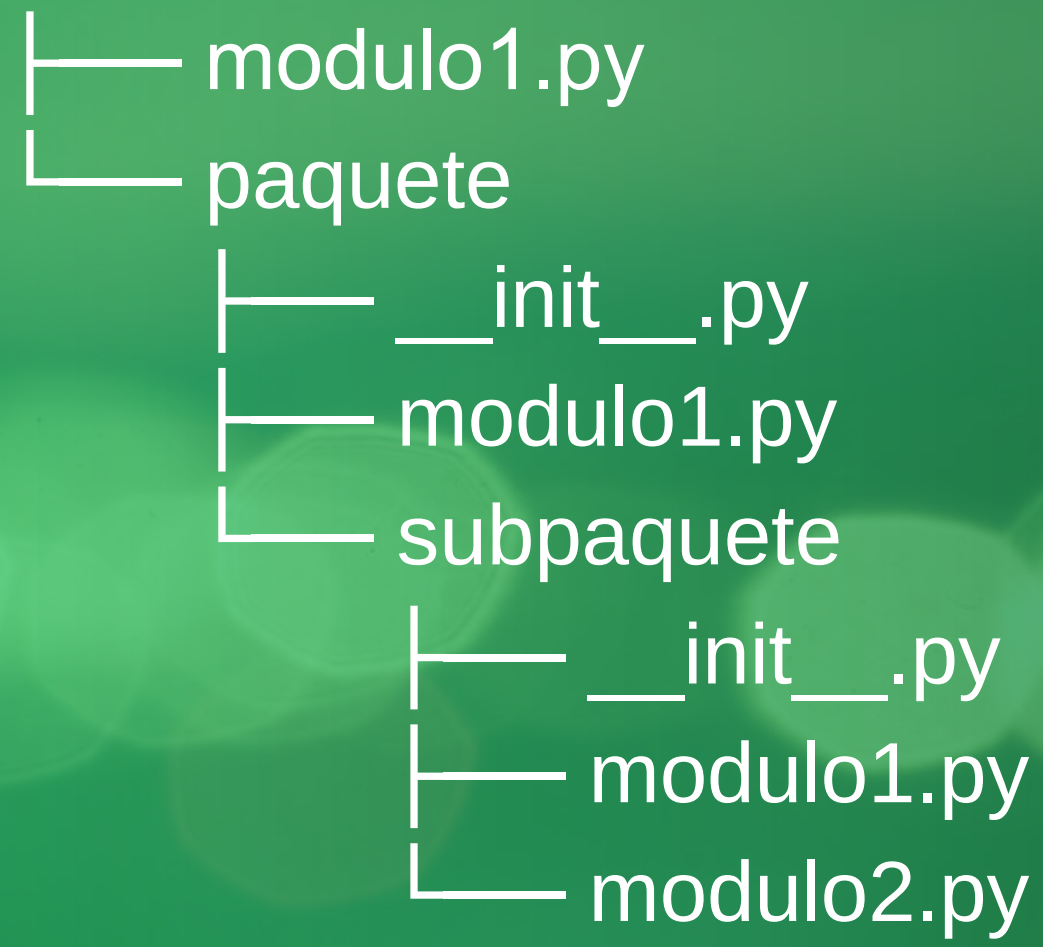
De hecho, puede estar completamente vacío.

```
└── Paquete
    ├── __init__.py
    ├── modulo1.py
    ├── modulo2.py
    └── modulo3.py
```

Los paquetes, a la vez, también pueden contener otros sub-paquetes:

```
└── paquete
    ├── __init__.py
    ├── modulo1.py
    └── subpaquete
        ├── __init__.py
        ├── modulo1.py
        └── modulo2.py
```

Y los módulos, no necesariamente, deben pertenecer a un paquete:



Trabajo con paquetes

```
import paquete.modulo1 as pm
import paquete.subpaquete.modulo1 as psm
```

Manejo de errores

Errores, fallas y otros bugs

Errores

Cualquier cosa que pueda salir mal, saldrá mal .

Esta es la ley de Murphy, y funciona en todas partes y siempre.

La ejecución de su código también puede salir mal. Si puede, lo hará.

Mira el código en el editor. Hay al menos dos formas posibles de que "salga mal". ¿Puedes verlos?

Como un usuario puede ingresar una cadena de caracteres completamente arbitraria, no hay garantía de que la cadena se pueda convertir en un valor flotante ; esta es la primera vulnerabilidad del código;

el segundo es que la sqrt()función falla si obtiene un argumento negativo .

Puede recibir uno de los siguientes mensajes de error.

Errores

Cualquier cosa que pueda salir mal, saldrá mal .

Esta es la ley de Murphy, y funciona en todas partes y siempre.

La ejecución de su código también puede salir mal. Si puede, lo hará.

Mira el código en el editor. Hay al menos dos formas posibles de que "salga mal". ¿Puedes verlos?

```
import math
x = float(input("Enter x: "))
y = math.sqrt(x)
print("The square root of", x, "equals to", y)
```

Exceptions / Excepciones

Cada vez que su código intenta hacer algo mal / tonto / irresponsable / loco / inaplicable, Python hace dos cosas:

1. **detiene su programa ;**
2. **crea un tipo especial de datos, llamado excepción .**

Ambas actividades se llaman plantear una excepción .

Podemos decir que Python siempre genera una excepción (o que se ha generado una excepción) cuando no tiene idea de qué hacer con su código.

¿Qué pasa después?

La excepción planteada espera que alguien o algo lo note y lo cuide;

Si no sucede nada para solucionar la excepción planteada, el programa se terminará por la fuerza y verá un mensaje de error enviado a la consola por Python.

De lo contrario, si la excepción se resuelve y se maneja adecuadamente, el programa suspendido se puede reanudar y su ejecución puede continuar.

Python proporciona herramientas efectivas que le permiten observar excepciones, identificarlas y manejarlas de manera eficiente.

Esto es posible debido al hecho de que todas las excepciones potenciales tienen sus nombres inequívocos, por lo que puede clasificarlos y reaccionar adecuadamente.

ZeroDivisionError

Mire el código en el editor. Ejecute el programa (obviamente incorrecto).

value = 1
value /= 0

Verá el siguiente mensaje en respuesta:

Traceback (most recent call last):
File "div.py", line 2, in
value /= 0
ZeroDivisionError: division by zero

Este error de excepción se llama ZeroDivisionError .

IndexError

Mire el código en el editor.

¿Qué pasará cuando lo ejecutes?

list = []

x = list[0]

Verá el siguiente mensaje en respuesta:

Traceback (most recent call last):

File "lst.py", line 2, in

x = list[0]

IndexError: list index out of range

¿Cómo manejas las excepciones?

La palabra **try** es clave para la solución.

- **try** también es una palabra clave.

La receta para el éxito es la siguiente:

1. **primero, tienes que intentar hacer algo ;**
2. **a continuación, debe verificar si todo salió bien.**

¿Cómo manejas las excepciones?

Pero, ¿no sería mejor verificar primero todas las circunstancias y luego hacer algo solo si es seguro?

Justo como el ejemplo en el editor.

```
firstNumber = int(input("Enter the first number: "))
secondNumber = int(input("Enter the second number: "))
if secondNumber != 0:
    print(firstNumber / secondNumber)
else:
    print("This operation cannot be done.")
print("THE END.")
```

¿Cómo manejas las excepciones?

Es cierto que esta forma puede parecer la más natural y comprensible, pero en realidad, este método no facilita la programación.

Todos estos controles pueden hacer que su código esté hinchado e ilegible .

Python prefiere un enfoque completamente diferente.

Mire el código en el editor. Le ayudará a comprender este mecanismo

```
try:
    print("1")
    x = 1 / 0
    print("2")
except:
    print("Oh dear, something went wrong...")

print("3")
```

Mire el código en el editor.

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
    print(y)
except ZeroDivisionError:
    print("You cannot divide by zero, sorry.")
except ValueError:
    print("You must enter an integer value.")
except:
    print("Oh dear, something went wrong...")
print("THE END.")
```

Si ingresa un valor entero válido distinto de cero (por ejemplo, 5) dice:

0.2

THE END.

si ingresa 0, dice:

You cannot divide by zero, sorry.

THE END.

si ingresa cualquier cadena no entera, verá:

You must enter an integer value.

THE END.

(localmente en su máquina) si presiona Ctrl-C mientras el programa está esperando la entrada del usuario (que causa una excepción llamada KeyboardInterrupt), el programa dice:

Oh dear, something went wrong...

THE END.

Notas

- **Las ramas except se buscan en el mismo orden en que aparecen en el código.**
- **No debe usar más de un, except con un cierto nombre de excepción;**
- **La cantidad de except diferentes es arbitraria: la única condición es que si usa try, debe colocar al menos una except (con o sin nombre);**
- **La palabra clave except no debe usarse sin un precedente try;**
- **Si except se ejecuta en alguna de las opciones, no se visitarán las otras;**
- **Si ninguna de las opciones de except especificadas coincide con la excepción planteada, la excepción no se controla**
- **Si existe una opción de except sin nombre, debe especificarse como la última.**

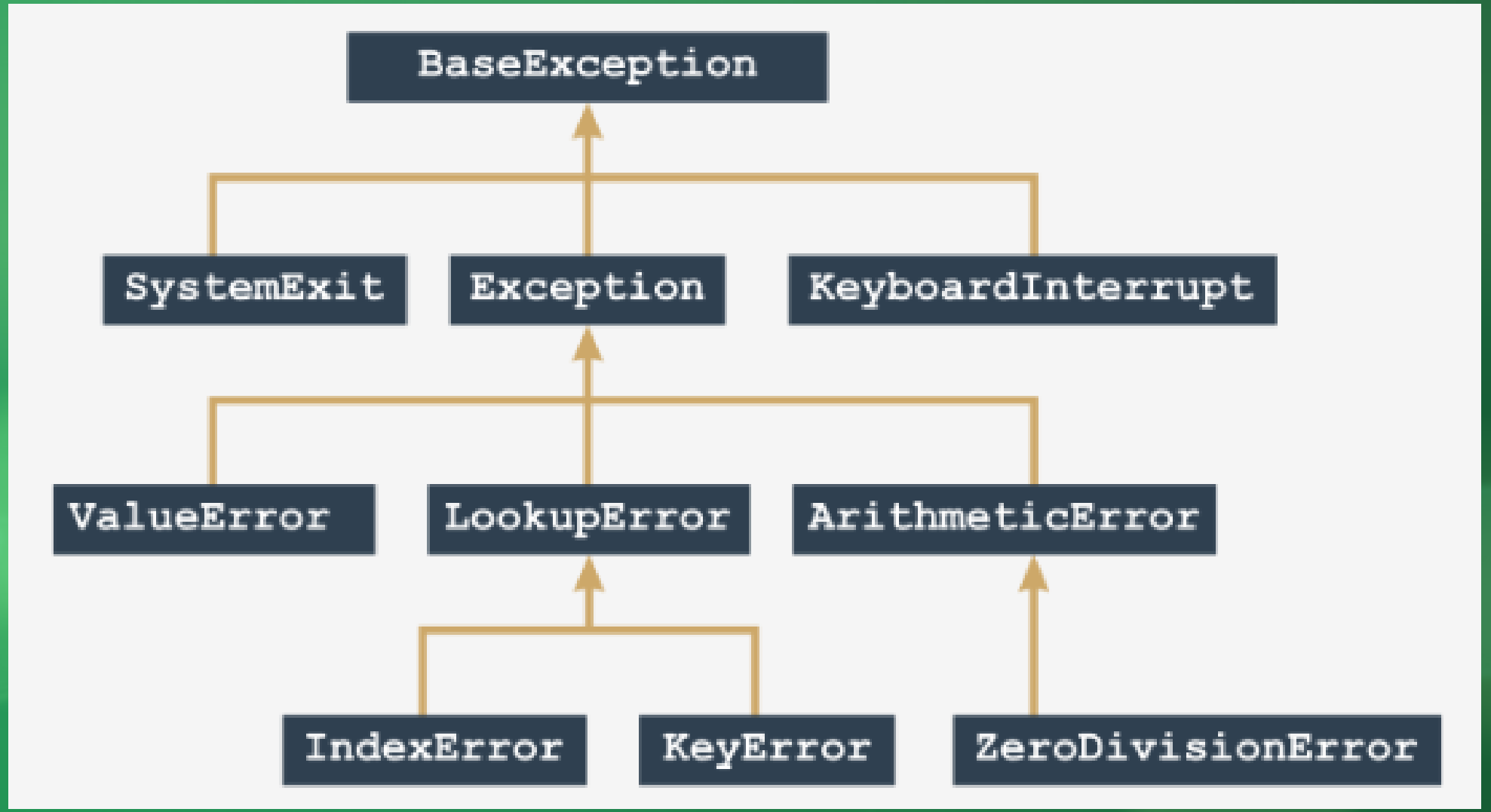
Excepciones

Python 3 define 63 excepciones incorporadas , y todas ellas forman una jerarquía en forma de árbol , aunque el árbol es un poco extraño ya que su raíz se encuentra en la parte superior.

Algunas de las excepciones incorporadas son más generales (incluyen otras excepciones) mientras que otras son completamente concretas (solo se representan a sí mismas).

Podemos decir que cuanto más cerca de la raíz se encuentra una excepción, más general es .

A su vez, las excepciones ubicadas en los extremos de las ramas (podemos llamarlas hojas) son concretas.


BaseException
SystemExit
Exception
KeyboardInterrupt
ValueError
LookupError
ArithmeticError
IndexError
KeyError
ZeroDivisionError

Ejemplos

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("Zero Division!")
except ArithmeticError:
    print("Arithmetic problem!")

print("THE END.")
```

```
try:
    y = 1 / 0
except ArithmeticError:
    print("Arithmetic problem!")
except ZeroDivisionError:
    print("Zero Division!")

print("THE END.")
```

Recuerde

1. ¡el orden de las ramas importa!
2. no ponga excepciones más generales antes que otras más concretas;

- esto hará que este último sea inalcanzable e inútil;
- además, hará que su código sea desordenado e inconsistente;
- Python no generará ningún mensaje de error con respecto a este problema.

- Si desea manejar dos o más excepciones de la misma manera, puede usar la siguiente sintaxis:

try:

    :

except (exc1, exc2):

    :

- Simplemente tiene que poner todos los nombres de excepciones comprometidos en una lista separada por comas y no olvidar los paréntesis.
- Si se genera una excepción dentro de una función , se puede manejar:

1. dentro de la función;
2. fuera de la función;

Variantes

Comencemos con la primera variante: mire el código en el editor.

```
def badFun(n):
    try:
        return 1 / n
    except ArithmeticError:
        print("Arithmetic Problem!")
    return None
badFun(0)
print("THE END.")
```

Variantes

- También es posible dejar que la excepción se propague fuera de la función
- Mire el código a continuación:

```
def badFun(n):
    return 1 / n
try:
    badFun(0)
except ArithmeticError:
    print("What happened? An exception was raised!")
print("THE END.")
```

Variantes

- El problema tiene que ser resuelto por el invocador (o por el invocador, etc.).
- El programa genera:
- What happened? An exception was raised!
- THE END.
- **Nota:** la excepción planteada puede cruzar la función y los límites del módulo , y viajar a través de la cadena de invocación buscando una except de coincidencia capaz de manejarlo.
- Si no existe tal cláusula, la excepción no se controla y Python resuelve el problema de la manera estándar: al finalizar su código y emitir un mensaje de diagnóstico.

Raise

La instrucción raise genera la excepción especificada denominada exc como si se hubiera generado de forma normal

raise exc

Nota: raise es una palabra clave.

La instrucción le permite:

1. **Simule excepciones reales (por ejemplo, para probar su estrategia de manejo)**
2. **Manejar parcialmente una excepción y hacer que otra parte del código sea responsable de completar el manejo.**

Miremos el script

```
def badFun(n):
    raise ZeroDivisionError

try:
    badFun(0)
except ArithmeticError:
    print("What happened? An error?")

print("THE END.")
```

La salida del programa permanece sin cambios.

De esta manera, puede probar su rutina de manejo de excepciones sin obligar al código a hacer cosas que no estén controladas.

La instrucción raise también se puede utilizar de la siguiente manera (tenga en cuenta la ausencia del nombre de la excepción):

Hay una restricción seria: este tipo de instrucción raise puede usarse solo dentro de la opción except ; usarlo en cualquier otro contexto causa un error.

La instrucción volverá a generar la misma excepción que se maneja actualmente.

Gracias a esto, puede distribuir el manejo de excepciones entre diferentes partes del código.

Mira el código en el editor.

```
def badFun(n):
    try:
        return n / 0
    except:
        print("I did it again!")
        raise
try:
    badFun(0)
except ArithmeticError:
    print("I see!")
print("THE END.")
```

El ZeroDivisionError se eleva dos veces:

1. primero, dentro de la parte try del código (esto es causado por la división cero)
2. segundo, dentro de except por la instrucción raise.

el código genera:

I did it again!

I see!

THE END.

assert

palabra clave assert

¿Como funciona?

Evalúa la expresión;

1. **si la expresión se evalúa como True, o un valor numérico distinto de cero, o una cadena no vacía, o cualquier otro valor diferente de None, no hará nada más;**
2. **de lo contrario, automáticamente e inmediatamente genera una excepción llamada AssertionError (en este caso, decimos que la afirmación ha fallado)**

Cómo usarlo?

- Es posible que desee ponerlo en su código donde quiera estar absolutamente a salvo de datos evidentemente incorrectos , y donde no esté totalmente seguro de que los datos hayan sido examinados cuidadosamente antes (por ejemplo, dentro de una función utilizada por otra persona).
- Generar una excepción AssertionError asegura que su código no produzca resultados no válidos y muestra claramente la naturaleza de la falla;
- assert no reemplazan las excepciones ni validan los datos, son sus complementos.
- Si las excepciones y la validación de datos son como conducir con cuidado, la afirmación puede desempeñar el papel de una bolsa de aire.

Mire el código

```
import math

x = float(input("Enter a number: "))
assert x >= 0.0

x = math.sqrt(x)

print(x)
```

Veamos las instrucciones assert en acción.

El programa se ejecuta sin problemas si ingresa un valor numérico válido mayor o igual a cero; de lo contrario, se detiene y emite el siguiente mensaje:

```
Traceback (most recent call last):
  File ".main.py", line 4, in
    assert x >= 0.0
AssertionError
```