exception handling

Ben Bolker

24 November 2019

generating errors

- we've already seen the `raise` keyword, in passing
- `raise Exception` is the simplest way to have your program stop when something goes wrong
- in a notebook/console environment, it stops the current cell/function (doesn't crash the session)

```
raise Exception
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception
```

- you have to `raise <something>`
- `Exception` is the most general case ("something happened")
- other possibilities
  - `TypeError`: some variable is the wrong type
  - `ValueError`: some variable is the right type but the wrong value

```
x = -1
if not isinstance(x,str): ## check if x is a str
    raise TypeError
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError
```

```
import math
x = -1
if x<0:
    raise ValueError
print(math.sqrt(x))
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError
```

error messages

- it's always better to be more specific about the cause of an error:

```
x = -1
if not isinstance(x,str): ## check if x is a str
    errstr = "x is of type "+type(x).__name__+", should be str"
    raise TypeError(errstr)
```

```
TypeError: x is of type int, should be str
```

f-strings are a convenient way to construct error messages: anything inside curly brackets is interpreted as a Python expression. e.g.

```
x=1
print(f"x is of type {type(x).__name__}, should be str")
```

```
## x is of type int, should be str
```

So we could use

```
if not isinstance(x,str): ## check if x is a str
    raise TypeError("x is of type {type(x).__name__}, should be str")
```

```
x = -1
if x<0:
    raise ValueError(f"x should be non-negative, but it equals {x}")
```

```
ValueError: x should be non-negative, but it equals -1
```

warnings

An error means "it's impossible to continue" or "you shouldn't continue without fixing the problem". You might want to issue a *warning* instead. This is not too different from just using `print()`, but it allows advanced users to decide if they want to suppress warnings.

```
import warnings
```

```
warnings.warn("something bad happened")
```

```
## <string>:1: UserWarning: something bad happened
```

handling errors

Now suppose you are getting an error and you don't want your program to stop. "Wrapping" your code in a `try:` clause will allow you to specify what to do in this case. `pass` is a special Python statement called a "null operation" or a "no-op"; it does nothing except keep going.

```
try:
    x= math.sqrt(-1)
except:
    pass
## keep going (but x will not be set)
```

You can specify something you want to do with only a particular set of errors:

```
try:
    x = math.sqrt(-1)
except ValueError:
    print("a ValueError occurred")
except:
    print("some other error occurred")
## keep going (but x will not be set)
```

```
## a ValueError occurred
```

If the error isn't caught because it isn't the right type, it will act like it normally does (without the `try:`)

```
try:
    z += 5  ## not defined yet
except ValueError:
    print("a ValueError occurred")
```

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

We could catch this with a general-purpose `except:`

```
try:
    z += 5  ## not defined yet
except ValueError:
    print("a ValueError occurred")
except:
    print("some other error occurred")
```

```
## some other error occurred
```

Or add another clause to catch it:

```
try:
    z += 5  ## not defined yet
except ValueError:
    print("a ValueError occurred")
except NameError:
    print("a NameError occurred")
except:
    print("some other error occurred")
```

```
## a NameError occurred
```

general rules

- see if you can change your code to avoid getting errors in the first place
- catch specific errors
- do something sensible with errors (e.g. convert to warnings, return `nan` ...)

```
try:
    x = math.sqrt(-1)
except ValueError:
    x = math.nan
print(x)
```

```
## nan
```