
goto; chicago
It really is easier to ask for forgiveness than permission
Naomi Ceder
follow us @gotochgo


goto;
chicago
Let us know what you think
Click ‘Rate Session’
to rate session
and ask questions.
follow us @gotochgo

Naomi Ceder, @naomiceder

- **Chair, Python Software Foundation**
- **Quick Python Book, 3rd ed**
- **Dick Blick Art Materials**

Introduction

- I'm a language nerd - human languages, that is (and computer languages, too)
- Every language has its way of doing things
  - It's not just vocabulary (Google Translate Sings)
  - the way of thinking about things/expressing things is different (examples)

Computer languages aren't as complex as human languages, but the same thing is true -

the structures of the language controls how you think about something.

What happens when things go wrong?

- Bad values
- Bad logic
- Unavailable resources
- Etc

You can also think of them as

- compile time - syntax errors, type errors (with static typing)
- run time - resource errors, errors from external processes, type errors (with dynamic typing)

or

- unrecoverable errors - syntax errors, type errors (with static typing)
- recoverable errors - resource errors, errors from external processes, type errors (with dynamic typing)

How does a language approach handling errors?

However you look at them, the approach a language takes to handling errors is an important part of how the language works;

it influences the structure and flow of the code.

perl - do or die

```
open(DATA, $file) || die "Error: Couldn't open the file $!";

die "Error: Can't change directory!: $!" unless(chdir("/etc"));
```

C

- return value

```
char *ptr = malloc(2000000000UL);
if (ptr == NULL) {
    perror("malloc failed");
```

- errno

```
fp = fopen("my_file.txt", "r");
printf(" Value of errno: %d\n ", errno);
```

- setjmp / longjmp
- segfault

C++

- Exceptions, but a lot of LBYL checking

In [ ]:

```
// Some code
cout << "Before try \n";
try {
   cout << "Inside try \n";
   if (x < 0)
   {
      throw x; // just simulating an error...
      cout << "After throw (Never executed) \n";
   }
}
catch (int x ) {
   cout << "Exception Caught \n";
}

cout << "After catch (Will be executed) \n";
return 0;
```

Java

- Exceptions, but a lot of LBYL checking
- "checked" (or "catch or specify") and "unchecked" exceptions

In [ ]:

```
public static void main(String[] args) {
     try {
         FileReader file = new FileReader("a.txt");
         BufferedReader fileInput = new BufferedReader(file);

         // Print 3 lines
         for (int counter = 0; counter < 3; counter++)
             System.out.println(fileInput.readLine());

         fileInput.close();
     }
      catch (IOException e) {
         System.err.println("Caught IOException: " + e.getMessage());
     }
 }
```

Javascript

- Exceptions (6 native types)
- But you can throw *anything*

In [ ]:

```
### Errors in Javascript

throw new Error();
throw true;

try{
    document.getElementById("mydiv").innerHTML='Success' //assuming "mydiv" is undefined
}
catch(e){
    if (e.name.toString() == "TypeError"){ //evals to true in this case
        //do something
    }
}
```

Go

- return result, error separately

In [ ]:

```
var err error
var a string
a, err = GetA()
if err == nil {
   var b string
   b, err = GetB(a)
   if err == nil {
      var c string
      c, err = GetC(b)
      if err == nil {
         return c, nil
      }
   }
}
return nil, err
```

They all have their advantages... and disadvantages...

And reflect the nature of the language.

What about Python?

Python's approach is to handle, rather than to avoid, errors

- EAFP - Easier to Ask Forgiveness than Permission
- contrast with, say, Java - LBYL "Look Before You Leap"

This approach makes sense for Python because...

- Simpler, easier to read code
- Duck typing
- Late binding of variables (types)

description of Python exceptions

```
try:
 Followed by block of code
```

```
except <Exception class> as e:
 Exception handling block
```

```
else:
 Block that executes if no exception is raised
```

```
finally:
 Block that is always executed, e.g., to close a file
```

You can also deliberately raise and exception: `raise <subclass of BaseException>`

In [2]:
```
try:
    print("running code")
    #raise Exception

except Exception as e:
    print("in exception block")

else:
    print("this executes if no exception")

finally:
    print("this always executes")
```

```
running code
this executes if no exception
this always executes
```

Exceptions and inheritance

- exceptions became classes in Python 1.5 (1997)
- only objects which are subclasses of `BaseException` can be raised (since Python 3)
- most exceptions are subclasses of `Exception`
- bare `except:` traps `Exception`
- `SystemExit`, `ExitGenerator`, and `KeyBoardInterrupt` inherit from `BaseException`, since they might not want to be trapped by a bare `except:`
- subclassing allows more precise catching of exceptions

In [3]:
```
raise Exception("Error occurred")
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-3-f31826022bcc> in <module>
----> 1 raise Exception("Error occurred")

Exception: Error occurred
```

In [ ]:

```
### Exception Class Hierarchy
"""
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
```

Sub-classing exceptions

- easy to have exceptions that specific to a module/library/package
- long, expensive, error prone, etc processes
- errors inside a chain of function calls and/or classes can be caught with more precision

In [7]:

```
## Custom (sub-classed) exceptions


class MySpecialException(Exception):
    pass


class MyEvenMoreSpecialException(MySpecialException):
    pass


try:
    #raise Exception("Exception")
    #raise MySpecialException("MySpecialException")
    raise MyEvenMoreSpecialException("MyEvenMoreSpecialException")
except MyEvenMoreSpecialException as e:
    print(e)
```

```
---------------------------------------------------------------------------
MySpecialException                        Traceback (most recent call last)
<ipython-input-7-6cd3c8bf67ea> in <module>
     12 try:
     13     #raise Exception("Exception")
---> 14     raise MySpecialException("MySpecialException")
     15     #raise MyEvenMoreSpecialException("MyEvenMoreSpecialException")
     16 except MyEvenMoreSpecialException as e:

MySpecialException: MySpecialException
```

Remember

- often one of the built in exceptions will do just as well a specific subclass
- go for the best trade off of readability/functionality
- if an exception will be thrown out of the module/library, the code handling it will need to import the exception

In [ ]:
```
# library specific exceptions
from my_library import SpecialClass, sub_library.ErrorOne, sub_library.ErrorTwo, sub_library
.ErrorThree
```

Observations

- Python has a very rich and well-developed system of exceptions
- Errors can be specific and handled according to inheritance hierarchy
- As an interpreted language, Python is suited to handle and recover from exceptions

Exceptions are more Pythonic than checking

Recommendations

in general, catching an exception is preferred to checking a result if:

- the exception is expected to be relatively infrequent
- the exception thrown will be identifiable and specific
- the code will be made easier to read...

In [ ]:

```
# Avoiding exceptions

for parameter in list_of_parameters:
    result = database.query_operation(parameter)
    if result is not None:
        print(result.count())
    else:
        continue
```

In [ ]:
```
 # with exceptions
for parameter in list_of_parameters:
    try:
        print(database.query_operation(parameter).count())
    except AttributeError:
        continue
```

Exception pitfalls

- bare excepts
- too many excepts
- code block too large
- poorly handled

Bare excepts

- Will not catch SystemExit, KeyboardInterrupt, or GeneratorExit (subclasses of BaseException)
- Will catch ALL subclasses of Exception , handle the same way
- Not Pythonic, rare to want to handle all possible exceptions with same code

In [ ]:

```
try:
    x = int(input("Enter an integer: "))
    print(10/x)
except:
    print("An error occurred...")
```

Too many excepts

- make code harder to read

In [ ]:

```
try:
    filename = input("Input filename: ")
except KeyBoardInterrupt as e:
    print("user interrupt")
    sys.exit()

try:
    for line in open(filename):
        try:
            value = float(line.strip())
        except ValueError as e:
            value = 0
        print(value)

except FileNotFoundError as e:
    # handle file not found
    print("File not found")
```

Code block too large

- difficult to handle specific errors
- location of error not specific

In [ ]:

```
""" Reads a file and returns the number of lines, words,
    and characters - similar to the UNIX wc utility
"""
import sys


def main():
    # initialze counts
    try:
        line_count = 0
        word_count = 0
        char_count = 0

        option = None
        params = sys.argv[1:]
        if len(params) > 1:
            # if more than one param, pop the first one as the option
            option = params.pop(0).lower().strip()
        filename = params[0]  # open the file
        with open(filename) as infile:
            for line in infile:
                line_count += 1
                char_count += len(line)
                words = line.split()
                word_count += len(words)

        if option == "-c":
            print("File has {} characters".format(char_count))
        elif option == "-w":
            print("File has {} words".format(word_count))
        elif option == "-l":
```

Poorly handled

- `pass` should be rare (maybe okay in debugging)

In [ ]:

```
try:
    filename = input("Input filename: ")
    for line in open(filename):
        value = float(line.strip())
        print(value)
except:
    pass
```

Guidelines for using exceptions

- Consider how often will the exception occur
- Be thoughtful about what exceptions you're handling and how
- Use built-in exceptions where it makes sense

Exceptions aren't just for errors any more...

Thanks to the Harry Potter Theory...

I'm sure that when J.K. Rowling wrote the first Harry Potter book (planning it as the first of a series of seven) she had developed a fairly good idea of what kind of things might eventually happen in the series, but she didn't have the complete plot lines for the remaining books worked out, nor did she have every detail decided of how magic works in her world.

I'm also assuming that as she wrote successive volumes, she occasionally went back to earlier books, picked out a detail that at the time was given just to add color (or should I say colour :-) to the story, and gave it new significance...

In a similar vein, I had never thought of iterators or generators when I came up with Python's for-loop, or using % as a string formatting operator, and as a matter of fact, using 'def' for defining both methods and functions was not part of the initial plan either (although I like it!).

~ Guido van Rossum, The Harry Potter Theory of Programming Language Design - https://www.artima.com/weblogs/viewpost.jsp?thread=123234 (https://www.artima.com/weblogs/viewpost.jsp?thread=123234)

Exceptions are raised by all of the following code snippets

How many of these are you aware of?

What exception(s) are raised?

In [ ]:
```
import sys

sys.exit(0)
```

In [ ]:
```
raise SystemExit(0)
```

SystemExit

- sys.exit() raises SystemExit exception
- raise SystemExit has the same effect

In [ ]:

```
a_list = [1, 2, 3, 4]

for i in a_list:
    print(i)
```

StopIteration

- Iterators raise a StopIteration exception to indicate that they are exhausted
- Some iterables with sequence semantics can raise an IndexError to tell the iterator that the end of sequence has been reached

In [8]:
```
for line in open("text_file.txt"):
    print(line)
```

```
line 1

line 2

line 3
```

EOFError

- Reading a file when there's nothing left to read raises an EOFError exception

In [9]:

```
def num_gen():
    numbers = [1, 2, 3, 4]
    for number in numbers:
        yield number
    print("Last number was sent")

for number in num_gen():
    print("Got", number)
    if number == 2:
        break

print("All done")
```

```
Got 1
Got 2
All done
```

In [11]:

```
def num_gen():
    numbers = [1, 2, 3, 4]
    try:
        for number in numbers:
            yield number
            print("Have sent", number)
    except GeneratorExit:
        print("GeneratorExit exception")
        # raise GeneratorExit

    print("Last number was sent")

#for number in num_gen():
#    print("Got", number)
#    if number == 2:
#        break
#print("Loop done")

gen1 = num_gen()
for number in gen1:
    print("Got", number)
    if number == 2:
        break
print("Loop done")

del gen1
```

```
Got 1
Have sent 1
Got 2
Loop done
```

GeneratorExit

- generators raise StopIteration when exhausted, like other iterators
- not "finishing" a generator object leaves it blocking after the yield latest yield...
- when the generator object is "finished", generator.close() raises a GeneratorExit exception at the last yield

In [13]:
```
class Foo:
    def __getattribute__(self, attr):
        try:
            print(f"About to get attribute {attr}")
            attr = super().__getattribute__(attr)
        except AttributeError as e:
            print(f"This class has no attribute {attr} - raising AttributeException")
            raise e
        return attr

    def __getattr__(self, attr):
        print(f"AttributeError raised when trying to get attribute {attr}")
        return f"You tried to get {attr}"


foo = Foo()
print(foo.__str__)
print(foo.bar)
```

```
About to get attribute __str__
<method-wrapper '__str__' of Foo object at 0x1071ae940>
About to get attribute bar
This class has no attribute bar - raising AttributeException
AttributeError raised when trying to get attribute bar
You tried to get bar
```

AttributeError

- if `__getattribute__` doesn't find an attribute name and raises an AttributeError...
- `__getattr__` is called and it should either compute/return the value or raise an AttributeError

What does all this mean?

In Python exceptions are used as form of flow control

- when the exception condition is expected to be very infrequent compared to the other conditions
- when the exception condition is rather different than the normal condition
- when using an exception instead of checking for the error condition makes code simpler

But using so many exceptions just feels... *wrong*...

- won't using a lot of exceptions hurt performance?
- doesn't using exceptions make the code more complex? harder to reason about? harder to test?

But... what about performance?

Aren't exceptions expensive?

Exceptions ARE a bit slower, but...

- they are optimized and are not as expensive as they were in, say, early C++
- they occur so rarely that there is little cost
- overall more Pythonic code tends to be faster

In [14]:

```
class Count():
    def __init__(self, count):
        self.count = count
    def __getitem__(self, key):
        if 0 < key < self.count:
            return key
        else:
            # IndexError raised to iterator
            raise IndexError

def test_count():
    counter = Count(1000)
    # iterator raises StopIteration to end interation
    for i in counter:
        x = i * i
```

In [17]:

```
def test_while_loop():
    i = 0
    length = 1000
    while i < length:
        x = i * i
        i += 1
```

In [15]:
```
%timeit test_count()
```

```
1.49 µs ± 102 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

In [18]:
```
%timeit test_while_loop()
```

```
218 µs ± 9.34 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Isn't using exceptions for flow control confusing/unreadable/somehow bad?

- Exceptions are such an integral part of Python, that by the time you notice, they should be understandable, Pythonic, even
- Used correctly they make the code more readable

Yes, (in Python) it really is easier to ask forgiveness than permission


goto;
chicago
Let us know what you think
Please
Remember to rate this session
Thank you!
follow us @gotochgo