Exceptions

Exceptions

- There are two approaches to living life as a religious:
  - Before you do anything, you ask for permission
    - Strengthens humility and denial of self
  - Do something and then ask for pardon
    - Strengthens your Ego too much, but makes it easier on the superior

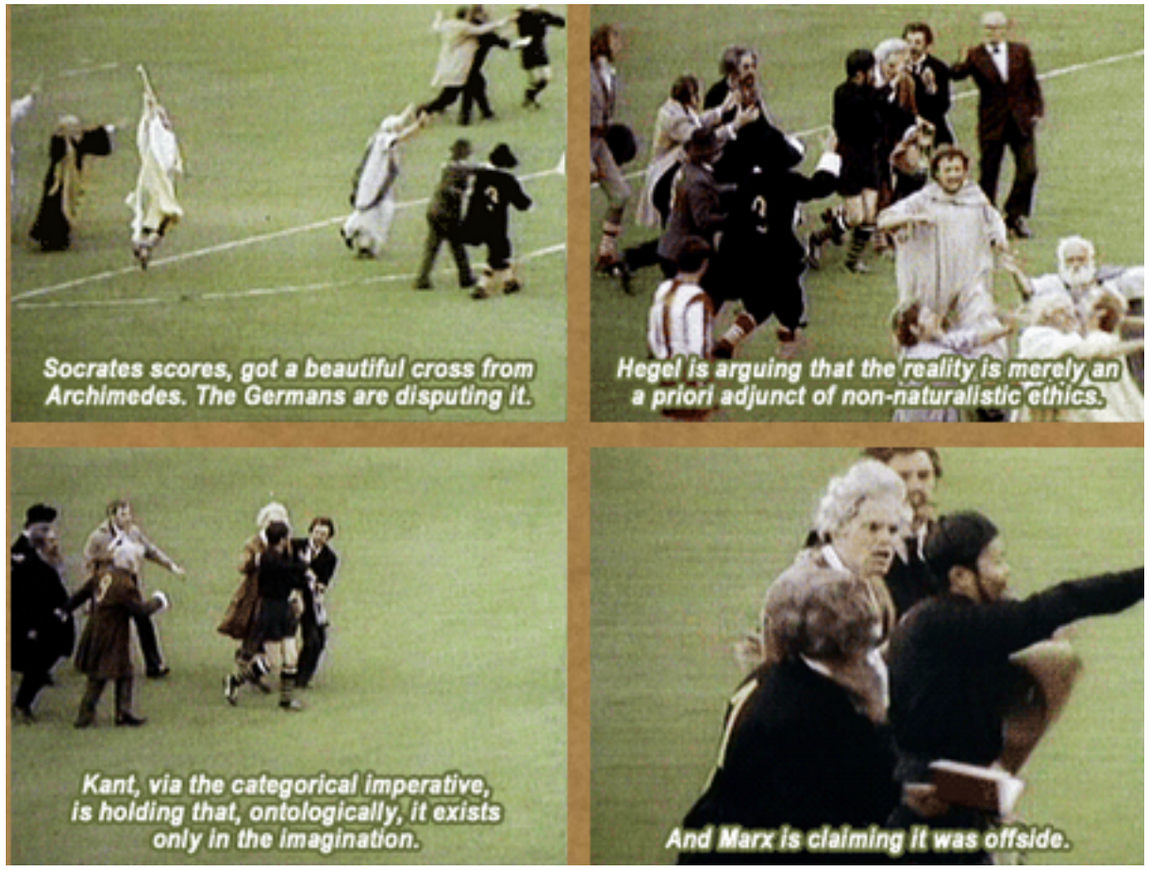
- Similarly: There are two approaches to the risks of live:
  - Make sure you are prepared for anything
  - Just live your life and deal with the consequences of your errors.

- In programming, Python tends to fall squarely into the second category
  - But it makes more sense than in real life

Exceptions

- *RAISING AN EXCEPTION* interrupts the flow of the program
- *HANDLING AN EXCEPTION* puts the program flow back on track or deals with an error situation
  - Such as out of memory, file cannot be found, CPU illegal instruction error, division by zero, overflow, …

Python Philosophy



Philosopher's Football

- Handle the common case.
  - And deal with the exceptions.

C, Java, C++ Philosophy

- C: check before you assume
- Java, C++: Use exceptions to handle bad situations
- Python: Use exceptions for the not so ordinary

Python

- If an instruction or block of instruction can cause an error, put it in a *try block*.

```
try:
    int(string)
```

Converts the string into an integer

```
Notice that we are not using the result of the conversion,
we just attempt the conversion
```

Python Exceptions

- Then afterwards, *handle the exception.*
  - You *should,* but are not required to specify the possible offending exception

```
try:
    int(string)
except ValueError:
    print("Conversion error")
```

If the conversion fails, a ValueError is thrown

This block handles the exception

Python Exceptions

- How do you find which error is thrown:
  - You can cause the error and see what type of error it is
  - You can look it up

```
>>> 5/0
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    5/0
ZeroDivisionError: division by zero
```

Division by zero creates a ZeroDivisionError

Python Exceptions

- Putting things together: Testing whether a string represents an integer

```
def is_int(string):
    try:
        int(string)
        return True
    except:
        return False
```

Try out the conversion

Python Exceptions

- Putting things together: Testing whether a string represents an integer

```
def is_int(string):
    try:
        int(string)
        return True
    except:
        return False
```

Try out the conversion

It worked:
We return True

Python Exceptions

- Putting things together: Testing whether a string represents an integer

```
def is_int(string):
    try:
        int(string)
        return True
    except:
        return False
```

Try out the conversion

It did NOT work:
An exception is thrown
We return FALSE

Python Exceptions

- As you can see from this example, the moment an exception is thrown, we jump to the exception handler.

Python Exceptions

- When to use exceptions and when to use if
  - Recall: Using `if` is defensive programming
  - Recall: Using exceptions amounts to the same degree of safety, but is offensive
- Rule of thumb:
  - If exceptions are raised infrequently, then use them

Python Exceptions

- Let’s make some timing experiments
  - Define two functions that square all elements in a list, if the elements are integers.

```
def square_list(lista):
    result = []
    for element in lista:
        if element.isdigit():
            result.append(int(element)**2)
```

```
def square_list2(lista):
    result = []
    for element in lista:
        try:

result.append(int(element)**2)
        except:
            pass
```

Python Exceptions

- The pass instruction:
  - When Python expects a statement, but we don't have one:
    - Just use `pass`
      - The No-Operation instruction

Python Exceptions

- Recall how to use the time-module to obtain the CPU (wall-clock) time
- We use this to measure execution time
  - First a list that only contains integers

```
def timeit(function, trials):
    lista = [str(i) for i in range(1000000)]
    count = 0
    for _ in range(trials):
        start = time.time()
        lista2 = function(lista)
        count += time.time()-start
    return count/trials
```

Python Exceptions

- Result: Exceptions are somewhat faster

```
>>> timeit(square_list, 5)
0.6882429599761963
>>> timeit(square_list2, 5)
0.615144681930542
```

Python Exceptions

- What if none of the list elements are integers:

```
def timeit(function, trials):
    lista = ["a"+str(i) for i in range(1000000)]
    count = 0
    for _ in range(trials):
        start = time.time()
        lista2 = function(lista)
        count += time.time()-start
    return count/trials
```

```
>>> timeit(square_list, 5)
0.07187228202819824
>>> timeit(square_list2, 5)
1.2984710693359376
```

Exceptions are much slower

Python Exceptions

- What about if the letter is at the end

```
def timeit(function, trials):
    lista = [str(i)+"a" for i in range(1000000)]
    count = 0
    for _ in range(trials):
        start = time.time()
        lista2 = function(lista)
        count += time.time()-start
    return count/trials
```

```
>>> timeit(square_list, 5)
0.09337239265441895
>>> timeit(square_list2, 5)
1.3271790504455567
```

Exceptions are still much slower

Self Test

- Define a function that calculates the geometric mean of two numbers.
- Use an exception to deal with a ValueError, arisen by taking the square-root of a negative number
- Here is the if-version. We return `None` if there is no mean.

```
def geo(x, y):
    if x*y > 0:
        return math.sqrt(x*y)
    return None
```

Self Test Solution

```
def geoe(x,y):
    try:
        return math.sqrt(x*y)
    except ValueError:
        return None
```

Multiple Exceptions

- We can write an exception handler that handles <u>all</u> the exceptions
  - This is discouraged since there are just too many exceptions that can occur
    - such as out-of-memory, system-error, keyboard-interrupt …
  - In this case, the except clause specifies no exception

```
try:
    accum += 1/n
except:
    print("something bad happened"
```

No exception specified
Handler handles everything

Multiple Exceptions

- Normally, you want to specify which exceptions you are handling
- You can specify several exception handles by repeating the exception clause
- Or you can handle a list of exceptions

The parentheses are necessary

```
def test():
    try:
        f = open("none.txt")
        block = f.read(256)
    except IOError:
        print("something happened when reading the file")
    except EOFError:
        print("ran out of file")
    except (KeyboardInterrupt, ValueError):
        print("something strange happened")
```

Cleaning Up

- Sometimes you need to make sure that failure-prone code cleans up
- Use the `finally` clause
  - Guaranteed to be executed
    - Even with return statements
    - Even when exceptions are raised

Example for `finally` clause

- If we open a file without the if-clause, we are morally obliged to close it
  - Let's say, if you have a long-running process that only needs a file for a little time, you should not hog the file and prevent others from accessing it.

Example for `finally` clause

```
def harmonic(filename):
    """
    Assumes that the elements in the file are numbers.
    We return the harmonic mean of the numbers.
    """
    count = 0
    accumulator = 0
    try:
        infile = open(filename, encoding="utf-8")
        for line in infile:
            for words in line.split():
                accumulator += 1/int(words)
                count += 1
        return count/accumulator
    except ZeroDivisionError:
        print("saw a zero")
        return 1000000000
    except ValueError:
        print("saw a non-integer")
        return 0
    finally:
        print("I am done and closing the file")
        infile.close()
```

Return in the try block

Return in the handler

But finally is guaranteed to run before any of the returns

Raising exceptions

- You can also raise your own exception
  - You can even define your own exceptions when you have understood classes
  - Just say: `raise ValueError`
  - or whatever the exception is that you want to raise.

Self Test

- Recall that the finally clause is always executed.
- What is the output of the following code

```
def raising():
    try:
        raise ValueError
    except ValueError:
        return 0
    finally:
        return 1
```

Answer

- The functions returns 1
  - The exception is raised and control passes to the exception handler
  - Before the exception handler can return, the finally clause is executed
  - And that one returns 1

Multiple Exceptions

- It is common that Python code throws multiple exceptions
  - Can list different exceptions using a tuple and handle them all

```
try:
   client_obj.get_url(url)
except (URLError, ValueError, SocketTimeout):
   client_obj.remove_url(url)
```

  - Or write different exception handlers

```
try:
   client_obj.get_url(url)
except (URLError, ValueError):
   client_obj.remove_url(url)
except SocketTimeout:
   client_obj.handle_url_timeout(url)
```

Handles to Exceptions

- Exceptions are classes that have methods
- To gain access use the `as` keyword

```
try:
   f = open(filename)
except OSError as e:
   if e.errno == errno.ENOENT:
      print('file not found')
   elif e.errno == errno.EACCES:
      print('permission denied')
   else:
      print('unexpected error')
```

Multiple Exceptions

- More than one exception can be triggered
  - The first matching exception handler will handle, even if a more specific exception handler is available

```
try:
   f = open(a_missing_file)
execpt OSError:
   print('it failed')
except FileNotFoundError:
   print('File not found')
```

  - prints out 'it failed'

Multiple Exceptions

- Exceptions are in a hierarchy

```
try:
    …
except Exception as e:
    …
    print(e)
```

  - catches all exceptions except SystemExit, KeyboardInterrupt, GeneratorExit
  - If you want to catch those, change Exception to BaseException

Creating Custom Exceptions

- To create a new exception, just define a class that derives from Exception

```
class NetworkError(Exception):
    pass
class TimeoutError(NetworkError):
    pass
```

Creating Custom Exceptions

- If your custom exception overrides the constructor
  - Make sure you call the exception class constructor

```
class CustomError(Exception):
    def __init__(self, message, status):
        self.message = message
        self.status = status
```

  - Parts of Python and libraries except all exceptions to have an .args attribute, that will be provided by calling the super

Chaining Exceptions

- Raise an exception in response to catching a different exception, but include information about both exceptions in the traceback

```
def example():
    try:
        int('N/A')
    except ValueError as e:
        raise RuntimeError('A parsing error occured') from e
```

Assertions

- To prevent error conditions, can use assertions
  - E.g.: your code only runs on a linux machine

```
import sys

assert ('linux' in sys.platform),
        'this code runs on linus only')
```

  - If the condition is violated, throws an AssertionError
  - But the assert statements are optimized away when

Else Statement

- Else block after a try block is executed only if no exception was raised

```
try:
```

run this code

```
except:
```

execute if there is an exception

```
else:
```

execute if there is **not** an exception

```
finally:
```

always run this code

-

Else Statement

- Exceptions in the else block would not be caught by the current try block

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Exercises

- The following code is potentially buggy.

```
info = [{'score': 3, 'confidence': 2},
        {'score': -1, 'confidence': 4},
        {'score': 1, 'confidence': 4},
        {'confidence': 0}]

def get_total_score(info):
   total = 0
   for item in info:
      total += item['score']
   return total

get_total_score(info)
```

Solutions

```
def get_total_score(info):
    total = 0
    number_of_items = 0
    for item in info:
        try:
            total += item['score']
        except KeyError:
            pass
        else:
            number_of_items += 1
    return total/number_of_items

print(get_total_score(info))
```

Exercises

- The following code is potentially buggy.

```
import os

def check(directory):
    for file_name in os.listdir(directory):
        with open(file_name) as infile:
            nr = len(infile.readlines())
            print(file_name, nr)
```

Solutions

```
import os

def check(directory):
    for file_name in os.listdir(directory):
        try:
            with open(file_name) as infile:
                nr = len(infile.readlines())
                print(file_name, nr)
        except UnicodeDecodeError:
            print('unicode decode error in', file_name)
        except IsADirectoryError:
            print(f'{file_name} is a directory')
```