Accelerating Information Technology Innovation

http://aiti.mit.edu

Ghana Summer 2011
Lecture 08 – Exceptions

MISTI

Do any of these look familiar to you?

```
SyntaxError: …
KeyError: …
IndexError: …
EOFError: …
IOError: …
AttributeError: …
ZeroDivisionError: …
NameError: …
TypeError: …
ValueError: …
```

Exceptional Situations

```
def calculate_infinity():
    infinity = 3/0
    return infinity
```

Exceptional Situations

```
def calculate_infinity():
    infinity = 3/0
    return infinity
```

Exception Terminology

- **Exceptions** are events that can modify the flow or control through a program.
- **try/except** : catch and recover from the error raised by you or the Python interpreter
- **finally**: perform cleanup actions whether exceptions occur or not
- **raise**: trigger an exception manually in your code
- **assert**: conditionally trigger an exception in your code

Dealing with Problems

Two Ways:

Look
Before
You
Leap

Easier to
Ask
Forgiveness than
Permission

Look Before You Leap

- Before we execute a statement, we check all aspects to make sure it executes correctly:
  - if it requires a string, validate it
  - if it requires a dictionary key, validate it

- Tends to make code messy. The heart of the code (what you want it to do) is hidden by all the checking.

Look Before You Leap

Example:

```
#LBYL, test for the problematic conditions
if not isinstance(s, str) or not s.isdigit:
    return None
elif len(s) > 10: # too many digits to
  convert
    return None
else:
    return int(str)
```

Easier to Ask Forgiveness than Permission

- Run any statement you want, no checking required.
- However, be ready to “clean up any messes” by catching errors that occur.
- The `try suite code` reflects what you want to do, and the `except code` what you want to do on error. Cleaner separation!
- **Python likes EAFP!**

Easier to Ask Forgiveness than Permission

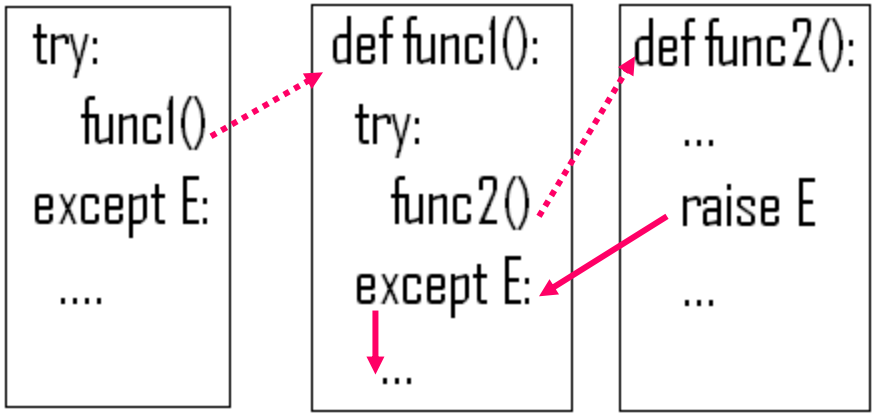
Example:

```
#EAFP, just do it, clean up messes
  with handlers
try:
    return int(str)
except (TypeError, ValueError,
  OverflowError):
    return None
```

Try, Except, Else and Finally

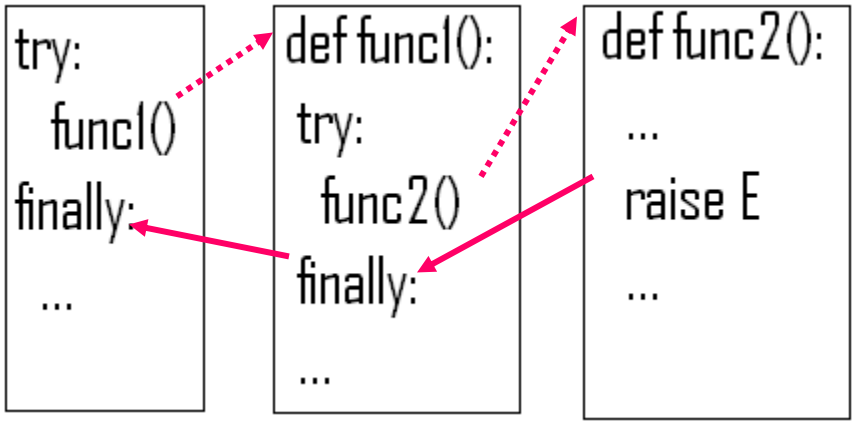
```
try:
    code to try

except pythonError1:
    exception code
except pythonError2:
    exception code
except:
     default except code

else:
    (No exceptions happened)


finally:
    clean up code
```

Nesting Exception Handlers



Once the exception is caught, its life is over.

Nesting Exception Handlers



- But if the ‘finally’ block is present the code in the finally block will be executed, whether an exception gets thrown or not.

User-Defined Exceptions

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
```

Raising exceptions

- We are running a bank, and don't allow people to have negative balances, so we have created a "NegativeBalanceError" exception.

```
if (balance-amount)<0:
    raise NegativeBalanceError
```

Exception Idioms

- All errors are exceptions, but not all exceptions are errors. It could be signals or warnings

```
>>>while line != “exit”:
        try:
            line=raw_input()
        except EOFError:
            break
        else:
            # process next line
```

- Functions signal conditions with *raise* (to distinguish success or failure)

Exception Design Tips

- Operations that commonly fail are generally wrapped in *try* statements. E.g:
  - file opens
  - socket calls
  - Database queries
- However, you may want failures of such operations to kill your program instead of being caught and ignored if the failure is a show-stopper. Failure = useful error message.
- Implement termination in *try/finally* to guarantee its execution.
- It is sometimes convenient to wrap the call to a large function in a single *try* statement rather than putting many *try* statements inside of the function.

Why try…except instead of If/else

- Someone else writes an API, but you are writing code on top of it. If there's an error, you need to know about it and handle it.
- If you don't catch exceptions, your program will die.
- Allows your program to recover from unexpected situations without writing code for every possible failure case

Applications/Common uses

- Databases (sql errors)
- Network communications (timeouts)
- Working with files (EOF, formats, corruption, file not found)
- Cameras
- Everything in Android (Android can throw 217 different exceptions!)