

COMP1730/COMP6730

Programming for Scientists

Exceptions and exception
handling



Lecture outline

- * The exception mechanism in python
- * Causing exceptions (`assert` and `raise`)
- * Handling exceptions

Reminder: Kinds of errors

- * Syntax errors: it's not python!
- * Runtime errors – code is syntactically valid, but you're asking the python interpreter to do something impossible.
 - E.g., apply operation to values of wrong type, call a function that is not defined, etc.
 - Causes an *exception*.
- * Semantic/logic errors: code runs without error, but does the wrong thing (for example, returns the wrong answer).

Exceptions

- * Exceptions are a control mechanism for handling runtime errors:
 - An exception is *raised* when the error occurs.
 - The exception moves up the call chain until it is *caught* by a *handler*.
 - If no handler catches the exception, it moves all the way up to the python interpreter, which prints an error message (and quits, if in script mode).
- * python allows the programmer to both raise and catch exceptions.

Exception names

- * Exceptions have *names*:
 - `TypeError`, `ValueError`
(incorrect type or value for operation)
 - `NameError`, `UnboundLocalError`,
`AttributeError`
(variable or function name not defined)
 - `IndexError` (invalid sequence index)
 - `KeyError` (key not in dictionary)
 - `ZeroDivisionError`
 - ...and others.

- * `https://docs.python.org/3/library/exceptions.html#concrete-exceptions` for full list of exceptions in python standard library.
- * Modules can define new exceptions.



Raising exceptions

Assertions - Recap

- * `assert condition, "fail message"`
 - Evaluate `condition` (to type `bool`)
 - If the value is not `True`, raise an `AssertionError` with the (optional) message.
- * Assertions are used to check the programmer's assumptions (including correct use of functions).
- * Function's docstring states assumptions; assertions can check them.

The `raise` statement

- * `raise ExceptionName(...)`
 - Raises the named exception.
 - Exception arguments (required or optional) depend on exception type.
- * Can be used to raise any type of runtime error.
- * Typically used with programmer-defined exception types.

Reminder: Defensive programming

- * Runtime errors are preferable to semantic errors, because it is immediately clear when and where they occur.
- * \Rightarrow it is better to “fail fast” (raise an exception) than to return a nonsense result.
- * Don't assert more than what is necessary.
 - For example, don't restrict types:

```
def fun(seq):  
    assert type(seq) == list  
    ...
```

is unnecessary if the function works for any sequence type.



Catching exceptions

Exception handling

```
try:  
    suite  
except ExceptionName:  
    error-handling suite
```

- * Execute *suite*.
- * If no exception arises, skip *error-handling suite* and continue as normal.
- * If the named exception arises from executing *suite* immediately execute *error-handling suite*, then continue as normal.
- * If any other error occurs, fail as normal.



- * An un-caught exception in a function causes an immediate end to the execution of the function suite; the exception passes to the function's caller, arising from the function call.
- * The exception stops at the *first* matching `except` clause encountered in the call chain.



* `f(2, -2), f("ab", "cd"), f("ab", 2):`
which error handler executes?

```
def f(x, y):  
    try:  
        return g(x, x + y)  
    except ZeroDivisionError:  
        return 0  
    except TypeError:  
        return 1
```

```
def g(x, y):  
    try:  
        return x / y  
    except TypeError:  
        return None
```

When to catch exceptions?

- * Never catch an exception unless there is a sensible way to handle it.
- * If a function does not raise an exception, its return value (or side effect) should be correct.
 - Therefore, if you can't compute a correct value, raise an exception!

Bad practice (delayed error)

```
def average(seq):  
    try:  
        return sum(seq) / len(seq)  
    except ZeroDivisionError:  
        print("empty sequence!")  
  
avg1 = average(a_seq)  
avg2 = average(b_seq)  
...  
if avg1 < avg2:  
    ...
```




* Repeat asking for input until valid:

```
number = None
while number is None:
    try:
        ans = input("Enter PIN:")
        number = int(ans)
    except ValueError:
        print("That's not a number!")
        number = None
```



- ★ Test if an operation is defined:

```
try:  
    n = len(seq)  
except TypeError:  
    n = 0 # type doesn't have length
```

- ★ A way to check if a value is “a sequence”, “iterable”, etc. (recall these are abstract concepts, not actual python types).
- ★ Few cases where this is useful.

```
try:  
    suite  
except ExceptionName:  
    error-handling suite  
finally:  
    clean-up suite
```

- ★ After *suite* finishes (whether it causes an exception or not), execute *clean-up suite*.
- ★ If an `except` clause is triggered, the error handler is executed before *clean-up suite*.
- ★ If the exception passes to the caller, *clean-up suite* is still executed before leaving the function.

★ Ensure file is closed even if an exception occurs:

```
def read_file(fname):  
    fo = open(fname)  
    try:  
        for line in fo:  
            # process line  
    finally:  
        fo.close() # close file
```

Summary

- * Consider:
 - What runtime errors can occur in your code?
 - Which should be caught, and how should they be handled?
 - What assumptions should be checked?
- * Use `assert` or `raise` to check violated assumptions.
- * Never catch an exception unless there is a sensible way to handle it.