



COMP1730/COMP6730

Programming for Scientists

Exceptions and exception handling



Lecture outline

- * **The exception mechanism in Python**
- * Raising exceptions (`assert` and `raise`)
- * Catching exceptions

Reminder: Kinds of errors

1. Syntax errors: it's not Python!
2. 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, division by zero, (large) etc.
 - Causes an **exception** (central concept in this lecture)
3. Semantic/logic errors: code runs without error, but does the wrong thing (for example, returns the wrong answer). Most severe, harder to detect errors

Exceptions

- * Exceptions are a built-in control mechanism in Python for systematically handling runtime errors:
 - An exception is **raised** when the runtime error occurs
 - *No further statements in the current code block are executed*
 - The exception moves up in the call stack until it is **caught** by an **exception 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 explicitly raise and catch exceptions (later in this lecture)

Exception names

- * Exceptions have **names**
- * Some examples of exception names built-in in Python:
 - `TypeError`, `ValueError`
(incorrect type or value for an operation or function)
 - `NameError` (variable or function name not defined)
 - `IndexError` (invalid sequence index)
 - `KeyError` (key not in dictionary)
 - `ZeroDivisionError`
 - and (many) others: click [here](#) for full list of built-in exceptions
- * Python can be extended with *custom* (i.e., programmer-defined) exception names (not covered in this lecture for simplicity)
- * For example, modules that you import may define new exceptions not necessarily in the Python standard library



Lecture outline

- * The exception mechanism in Python
- * **Raising exceptions (`assert` and `raise`)**
- * Catching exceptions

Assertions: the `assert` statement

- * `assert condition, "fail message"`
 - Evaluates `condition`
 - If the value of `condition` is not `True`, raises an `AssertionError` along with the message
 - (Message is optional)
- * Assertions are a very useful mechanism to explicitly check the programmer's assumptions, e.g., on function arguments
- * Function's doc-string states assumptions; assertions explicitly check them
- * We have also used assertions thoroughly in test functions as a mechanism to detect semantic errors (i.e., to check for code correctness)

Raising exceptions: the `raise` statement

- * `raise ExceptionName(...)`
 - Raises the named exception
 - Exception arguments (required and optional) depend on exception type
- * Can be used to raise any type of runtime error
- * Typically used to raise programmer-defined exception types (although not necessarily, as shown in the example below)
- * What is the difference among these two Python codes?

```
assert type(var) == list, 'var is not a list'
```

```
if type(var) != list:  
    raise TypeError('var is not a list')
```

Reminder: Defensive programming

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

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

is unnecessary if the function works for any sequence type



Lecture outline

- * The exception mechanism in Python
- * Raising exceptions (`assert` and `raise`)
- * **Catching exceptions**

Exception handling

```
try:
    suite                                # May contain several instructions
except ExceptionName:
    error-handling suite # May contain several instructions
```

- * (Try to) Execute the instructions within *suite*
- * If no exception arises while executing *suite*, skip *error-handling suite* and continue as normal
- * If *ExceptionName* arises, jump to *error-handling suite*, then continue with instructions below *try-except* clause
- * If any other exception different from *ExceptionName* arises, handle it as if no *try-except* clause was present (next slide)
- * **NOTE:** there can be more than one *except :* clause in the same *try-except* statement (thus allowing to catch and handle different exceptions in a different way)
- * **NOTE:** *ExceptionName* can be omitted from *except :* (thus allowing to catch and handle any exception the same way)

Exception handling and functions

- * An exception raised in a function interrupts the execution of the function suite (i.e., remaining instructions are skipped)
- * If the exception is caught by a `try-except` statement, then the error handling suite is executed (as seen in previous slide)
- * BUT, if the exception is uncaught, then it is moved up to the function's caller
- * The exception stops being moved up in the call chain at the **first** matching `except` clause encountered in the call chain

Exception handling and functions (Example)

```
def g(x, y):  
    try:  
        return x / y  
    except TypeError:  
        return None  
  
def f(x, y):  
    try:  
        return g(x, x + y)  
    except ZeroDivisionError:  
        return 0  
    except TypeError:  
        return 1
```

Which error handler suite executes? Which value do the following function calls return?

- * `f(2, -2)`
- * `f("ab", "cd")`
- * `f("ab", 2)`

When to catch exceptions?

- ★ Never catch an exception unless there is a sensible way to handle it
- ★ If a function call does not raise an exception, its return value (or side effect in the case of functions modifying arguments) should be correct
- ★ Therefore, if you cannot compute a correct value, raise an exception to the caller

Bad practice example (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:  
    ...
```

- * Exception caught but not handled properly
- * What happens, e.g., if `a_seq` is empty but `b_seq` is not?
- * Violation of fail-fast principle



Good practice example

```
def input_number():  
    """Input a number from keyboard with error checking"""  
    number = None  
    while number is None:  
        try:  
            ans = input("Enter PIN:")  
            number = int(ans)  
        except ValueError:  
            print("That's not a number!")  
            number = None  
    return number
```

Keep asking for keyboard input until input is valid



Another good practice example

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

- * Test if an operation (e.g., `len`) is defined on a given object
- * 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 actually useful, though



try-except-finally

```
try:
    suite                # May contain several instructions
except ExceptionName:
    error-handling suite # May contain several instructions
finally:
    clean-up suite      # May contain several instructions
```

- * 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



try-except-finally (Example)

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

Ensure file will be closed even if an exception occurs

Take-home messages

- * Systematically consider:
 - What runtime errors can potentially occur in your code?
 - Which should be caught, and how should they be handled?
 - What assumptions should be checked?
- * Use `assert` or `raise` to explicitly check on assumptions
- * Never catch an exception if you do not know how to handle it
- * Use exceptions to systematically treat runtime errors as apposed to, e.g.. `if+print(error_message)+exit()` statements scattered accross the code (this would prevent the caller to deal with the exception in a different way)