

COMP1730/COMP6730 Programming for Scientists

Exceptions and exception handling



Lecture outline

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



Types of errors

- Syntax errors: evident as soon as you try to run the code.
- Runtime errors: arise when the code runs (and maybe only under certain conditions).
 - Applying a function or operator to the wrong value, or wrong type of value;
 - Indexing past the beginning/end of a list;
 - and many more.
- Semantic 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



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

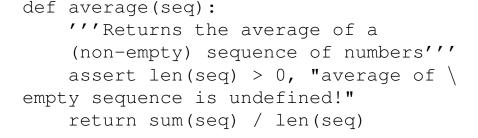


Raising exceptions



Assertions

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







Why assert?

- "Fail fast": it is usually better for a function to raise an exception as soon as a violation of assumptions is detected.
- * Provide specific error information.
 - "average of empty sequence is undefined" is more explanatory than *ZeroDivisionError*
- It is *always* better to raise an exception than return an incorrect (garbage) result.
- * Semantic errors are the hardest to find!



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.



Examples

- What assumptions can or should be checked in our implementations of
 - the recursive/iterative interval-halving algorithm;
 - finding the greatest element ≤ x in a sorted sequence;
 - the "network" or "grid" ADTs?
- * What error should be raised if they do not hold?



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 errorhandling suite, then continue as normal.
- * If any other error occurs, fail as normal.



* 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.



- An un-caught exeception 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 q(x, x + y)
    except ZeroDivisionError:
        return 0
    except TypeError:
        return 1
def q(x, y):
    try:
        return x / y
    except TypeError:
        return None
```



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:



Summary

- Never catch an exception unless there is a sensible way to handle it.
- If a function does not raise an exception, it's return value (or side effect) should be correct.
 - Therefore, if you can't compute a correct value, raise an exception!
- * Consider:
 - What runtime errors may occur?
 - Which should be caught, and how should they be handled?
 - What assumptions should be checked?