Python 3 Exceptions & Handling Errors

Eddie Guo

October 2019

1 Introduction to Exceptions

1.1 Topics Covered

(i) What are exceptions?

(ii) Raising exceptions

(iii) Catching exceptions

(iv) Assertions

1.2 What are Exceptions?

- Input validation != exception handling
- Exception: event during execution of program that disrupts normal flow of program
- Exceptions allow us to handle errors/exceptional conditions
- In Python, exception is obj that reps an error

1.3 Exceptions in Python

- Python script raises an exception where error detected
- Python interpreter raises exception when it detects run-time error
- Can explicitly raise an exception
  - '2013' + 1 → **TypeError**: cannot concatenate 'str' and 'int' objects
  - 175 + cmput*13 → **NameError**: name 'cmput' is not defined
  - 365 * (12/0) → **ZeroDivisionError**: integer division or modulao by zero
- Accessing non-existent dictionary key will raise **KeyError** exception
- Searching list for non-existent value will raise **ValueError** exception
- Calling non-existent method will raise **AttributeError** exceptiont
- Python documentation for exceptions

1.4 Why Use Exceptions?

- Separating error-handling code from regular code
- Deferring decisions about how to respond to exceptions
- Providing mech for specifying diff kinds of exceptions that can arise in program

1.5 Exception Handling Blocks

- If you have code that may raise exception, place code in `try:` followed by `except:`
- Don't catch exception? Entire program crashes.
- `except` w/o explicit exception will catch all remaining exceptions
- `except` may name none/one/multiple exceptions as parenthesized tuple

```
except (RuntimeError, TypeError,
    NameError):
        [do something here]
```

1.6 Multiple Except Clauses

- `try` may have >1 `except` clause to specify handlers for diff exceptions
- At most, one handler will be executed
- Handlers only handle exceptions that occur in corresponding try clause, not in other handlers of same try statement
- Go from specific excpetions to more general ones b/c Python reads top-down

1.7 The `try` Statement

- If no exception raised by code w/in try block (or methods called w/in try block), code executes normally & all except blocks skipped
- If exception arises in try block, execution of try block terminates execution immediately & `except` is sought to handle exception
  1. If appropriate except clause found, it's executed
  2. Elif exception propagated to method or outer `try` block
  3. Elif no handler found → **unhandled exception** & execution stops w/ message

1.8 Propagating Exceptions

- An exception will bubble up call stack until it:
  - Reaches method w/ suitable handler or
  - Propagates thru main stack (1st method on call stack)
- If exception not caught by any method, exception treated like error: stack frames displayed & program terminates

```
try:
    f = open('myfile.txt', 'r')
    s = f.readline()
    i = int(s.strip())
except IOError:
    print('File does not exist or cannot be read.')
except ValueError:
    print('Could not convert data to an integer')
except: # If this were first, no IOErrors or ValueErrors will be caught
    print('Unexpected error')
    raise # can explicitly propagate exceptions using raise
```

1.9 Raising Exceptions

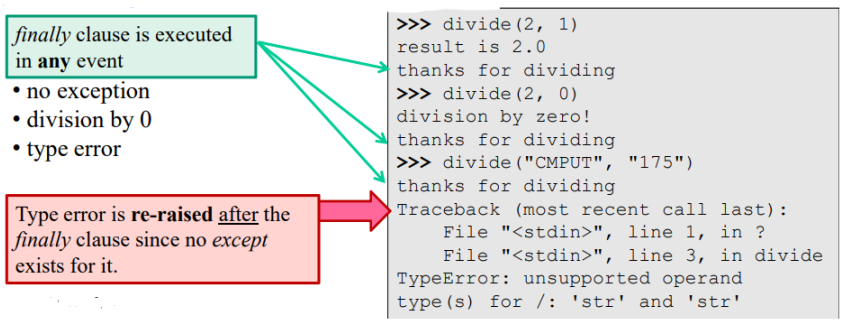
- What can be raised as exception?
  - Any standard Python exception
  - New instance of exception w/ custom arguments
  - Instances of our own specialized exception classes

```
try:
    print('Raising an exception')
    raise Exception('CMPUT', '274')
except Exception as inst: # the exception instance
    print(inst.args) # arguments stored in .args
    x, y = inst.args # unpacks args
    print('x =', x, 'y =', 'y')
```

1.10 `else` & `finally` Clause

- `else` clause CAN'T come b4 try & except (i.e., must follow all except clauses)
- Code in else clause must be executed if clause does not raise exception
- `finally` will execute regardless if error was raised (executed under ALL circumstances)
- `finally` useful if you wanna perform "cleanup" operations b4 exiting method (ex: closing file) & avoids duplicating code in each `except` clause

```python
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print('division by zero!')
    else:
        print('the result is', result)
    finally:
        print('thanks for dividing!')
```



2 Summary: Possible Execution Paths

1. No exception occurs
   (a) Execute try block
   (b) Execute else & finally clauses
   (c) Execute rest of method
2. Exception occurs & is caught
   (a) Execute try block until 1st exception occurs
   (b) Execute 1st except clause that matches exception
   (c) Execute finally clause
   (d) Execute rest of method
3. Exception occurs & is not caught
   (a) Execute try block until 1st exception occurs
   (b) Execute try block until 1st exception occurs
   (c) Execute finally clause
   (d) Propagate exception to calling method

3 Assertions

- Assertion is statement that raises **AssertionError** exception if condition not met
- `assert Expression[, Arguments]`
- If assertion fails, Python uses given arg as arg for `AssertionError`
- `AssertionError` exceptions can be caught & handled like any other exception
- Good practice to place assertions at start of fn to check for valid input, & after fn call to check for valid output

```python
def KelvintoFahrenheit(temperature):
    assert (temperature >= 0), 'Colder than absolute zero!'
    return ((temperature - 273) * 1.80 + 32

if __name__ == '__main__':
    try:
        fahrenheit = KelvintoFahrenheit(-23)
        print(fahrenheit)
    except AssertionError as my_error:
        print(my_error.args)

# Output
(Colder than absolute zero!, )
```