Programming Principles in Python (CSCI 503)

Debugging & Testing

Dr. David Koop
Dealing with Errors

• Can explicitly check for errors at each step
  - Check for division by zero
  - Check for invalid parameter value (e.g. string instead of int)

• Sometimes all of this gets in the way and can't be addressed succinctly
  - Too many potential errors to check
  - Cannot handle groups of the same type of errors together

• Allow programmer to determine when and how to handle issues
  - Allow things to go wrong and handle them instead
  - Allow errors to be propagated and addressed once
Advantages of Exceptions

• Separate error-handling code from "regular" code
• Allows propagation of errors up the call stack
• Errors can be grouped and differentiated
Try-Except

• The `try` statement has the following form:

```python
try:
    <body>
except <ErrorType>*:
    <handler>
```

• When Python encounters a `try` statement, it attempts to execute the statements inside the body.

• If there is no error, control passes to the next statement after the `try...except` (unless `else` or `finally` clauses)

• Note: `except` not catch
Exception Granularity

- If you catch any exception using a base class near the top of the hierarchy, you may be masking code errors

```python
try:
    c, d = a / b
except Exception:
    c, d = 0, 0
```

- Remember `Exception` catches any exception is an instance of `Exception`
- Catches `TypeError: cannot unpack non-iterable float object`
- Better to have more granular (specific) exceptions!
- We don't want to catch the `TypeError` because this is a programming error not a runtime error
Exception Locality

- try:
  
  ```python
  fname = 'missing-file.dat'
  with open(fname) as f:
      lines = f.readlines()
  except OSError:
      print(f"An error occurred reading {fname}\")
  
  try:
      out_fname = 'output-file.dat'
      with open('output-file.dat', 'w') as fout:
          fout.write("Testing")
  except OSError:
      print(f"An error occurred writing {out_fname}\")
  ```
Multiple Except Clauses

• Function like an if/elif sequence
• Checked in order so put more granular exceptions earlier!

```python
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except FileNotFoundError:
    print(f"File {fname} does not exist")
except OSError:
    print("An error occurred processing files")
```
Handling Multiple Exceptions at Once

- Can process multiple exceptions with one clause, use tuple of classes
- Allows some specificity but without repeating

```python
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except (FileNotFoundError, PermissionError):
    print("An error occurred processing files")
```
Exception Objects

- Exceptions themselves are a type of object.
- If you follow the error type with an identifier in an except clause, Python will assign that identifier the actual exception object.
- Sometimes exceptions encode information that is useful for handling

```python
try:
   .fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except OSError as e:
    print(e.errno, e.filename, e)
```
Else & Finally

- else: Code that executes if no exception occurs
- finally: Code that always runs, **regardless** of whether there is an exception

```python
b = 3
a = 0
try:
    c = b / a
except ZeroDivisionError:
    print("Division failed")
    c = 0
else:
    print("Division succeeded", c)
finally:
    print("This always runs")
```
Raising Exceptions

• Create an exception and raise it using the `raise` keyword
• Pass a string that provides some detail
• Example: `raise Exception("This did not work correctly")`
• Try to find an exception class:
  - `ValueError`: if an argument doesn't fit the functions expectations
  - `NotImplementedError`: if a method isn't implemented (e.g. abstract cls)
• Be specific in the error message, state actual values
• Can also subclass from existing exception class, but check if existing exception works first
• Some packages create their own base exception class (`RequestException`)
Making Sense of Exceptions

• When code (e.g. a cell) crashes, read the traceback:

  ZeroDivisionError: division by zero

  Traceback (most recent call last):
  <ipython-input-58-488e97ad7d74> in <module>
    4     return divide(a+b, a-b)
    5   for i in range(4):
----> 6   process(3, i)
  <ipython-input-58-488e97ad7d74> in process(a, b)
    3        return c / d
----> 4   return divide(a+b, a-b)
    5   for i in range(4):
  <ipython-input-58-488e97ad7d74> in divide(c, d)
    2    def divide(c, d):
----> 3    return c / d
    4    return divide(a+b, a-b)

D. Koop, CSCI 503, Spring 2021
Assignment 6

• Object-oriented Programming
• Track University Enrollment
• Methods for checking conflicts (e.g. disallow student to have overlapping courses, take too many credits)
• Methods for changing course time (check the new time works for everyone)
• Sample code is meant to be run in different cells!
• Due Friday
How do you debug code?
Debugging

- print statements
- logging library
- pdb
- Extensions for IDEs (e.g. PyCharm)
- JupyterLab Debugger Support
Print Statements

• Just print the values or other information about identifiers:
  ```python
def my_function(a, b):
    print(a, b)
    print(b - a == 0)
    return a + b
  ```

• Note that we need to remember what is being printed
• Can add this to print call, or use f-strings with trailing = which causes the name and value of the variable to be printed
  ```python
def my_function(a, b):
    print(f"{a=} {b=} {b - a == 0}")
    return a + b
  ```
Print Problems

• Have to uncomment/comment
• Have to remember to get rid of (or comment out) debugging statements when publishing code
• Print can dump a lot of text (slows down notebooks)
• Can try to be smarter:
  - if i % 100 == 0:
    print(i, f"{current_output=}")
  - do_print = value == 42
    if do_print:
      print(f"{a=} {current_output=}")
Logging Library

- Allows different levels of output (e.g. DEBUG, INFO, WARNING, ERROR, CRITICAL)
- Can output to a file as well as stdout/stderr
- Can configure to suppress certain levels or filter messages

```python
import logging
def my_function(a, b):
    logging.debug(f"{a=} {b=} {b-a == 0}")
    return a + b
my_function(3, 5)
```

- This doesn't work in notebooks…
Logging Library

• Need to set default level (e.g. DEBUG)
• For notebooks, best to define own logger and set level
• `import logging
  logger = logging.Logger('my-logger')
  logger.setLevel(logging.DEBUG)
  def my_function(a,b):
    logger.debug(f"{a=} {b=} {b-a == 0}")
    return a + b
  my_function(3, 5)

• Prints on stderr, can set to stdout via:
• `import sys
  logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)`
Python Debugger (pdb)

- Debuggers offer the ability to inspect and interact with code as it is running
  - Define breakpoints as places to stop code and enter the debugger
  - Commands to inspect variables and step through code
  - Different types of steps (into, over, continue)
  - Can have multiple breakpoints in a piece of code
- There are a number of debuggers like those built into IDEs (e.g. PyCharm)
- pdb is standard Python, also an ipdb variant for IPython/notebooks
Python Debugger

• Post-mortem inspection:
  - In the notebook, use `%%debug` in a new cell to inspect at the line that raised the exception
  • Can have this happen all the time using `%%pdb` magic
  • Brings up a new panel that allows debugging interactions
  
- In a script, run the script using `pdb`:
  • `python -m pdb my_script.py`
Python Debugger

- **Breakpoints**
  - To set a breakpoint, simply add a `breakpoint()` call in the code
  - Before Python 3.7, this required `import pdb; pdb.set_trace()`
  - Run the cell/script as normal and pdb will start when it hits the breakpoint

```python
> <ipython-input-1-792bb5fe2598>(3)divide()
 1 def process(a, b):
 2   def divide(c, d):
 3     return c / d
 4   return divide(a+b, a-b)
 5 result = []

ipdb>
```
Python Debugger Commands

- `p` [print expressions]: Print expressions, comma separated
- `n` [step over]: continue until next line in **current function**
- `s` [step into]: stop at next line of code (same function or one being called)
- `c` [continue]: continue execution until next breakpoint
- `l` [list code]: list source code (ipdb does this already), also `ll` (fewer lines)
- `b` [breakpoints]: list or set new breakpoint (with line number)
- `w` [print stack trace]: Prints the stack (like what notebook shows during traceback), `u` and `d` commands move up/down the stack
- `q` [quit]: quit
- `h` [help]: help (there are many other commands)
Jupyter Debugging Support

```
[ ]: 1
def add(a, b):
    res = a + b
    return res

[ ]: 1
    res = add(1, 2)
    res += 1
    res
```
Jupyter Debugging Support
How do you test code?
Testing

- If statements
- Assert statements
- Unit Testing
- Integration Testing
Testing via Print/If Statements

• Can make sure that types or values satisfy expectations
  • if not isinstance(a, str):
    raise Exception("a is not a string")
  • if 3 < a <= 7:
    raise Exception("a should not be in (3,7]")
• These may not be something we need to always check during runtime
Assertions

- Shortcut for the manual if statements
- Have python throw an exception if a particular condition is not met
- `assert` is a keyword, part of a statement, not a function
- `assert a == 1, "a is not 1"
- Raises `AssertionError` if the condition is not met, otherwise continues
- Can be caught in an except clause or made to crash the code
- Problem: first failure ends error checks
Unit Tests

• "Testing shows the presence, not the absence of bugs", E. Dijkstra
• Want to test many parts of the code
• Try to cover different functions that may or may not be called
• Write functions that test code

```python
def add(a, b):
    return a + b + 1

def test_add():
    assert add(3, 4) == 7, "add not working"

def test_operator():
    assert operator.add(3, 4) == 7, "__add__ not working"
```

• If we just call these in a program, first error stops all testing
Unit Testing Framework

- unittest: built in to Python Standard Library
- nose2: nose tests, was nose, now nose2 (some nicer filtering options)
- pytest: extra features like restarting tests from last failed test
- doctest: built-in, allows test specification in docstrings

With the exception of doctest, the frameworks allow the same specification of tests
unittest

- Subclass from unittest.TestCase, write test_* functions
- Use assert* instance functions
- import unittest

```python
class TestOperators(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(3, 4), 7)

    def test_add_op(self):
        self.assertEqual(operator.add(3, 4), 7)

unittest.main(argv=[''], exit=False)
```
Lots of Assertions

- `assertEqual/assertNotEqual`: smart about lists/tuples/etc.
- `assertLess/assertGreater/assertLessEqual/assertGreaterEqual`
- `assertAlmostEqual`: allows for floating-point arithmetic errors
- `assertTrue/assertFalse`: check boolean assertions
- `assertIsNone`: check for `None` values
- `assertIn`: check containment
- `assertIsInstance`
- `assertRegex`: check that a regex matches
- `assertRaises`: check that a particular exception is raised
Test Options

- Run only certain tests
  - `argv=[]` # run default set of tests
  - `argv=['', 'TestLists']` # run all test* methods in TestLists
  - `argv=['', 'TestAdd.test_add']` # run test_add in TestAdd

- Show more detailed output
  - By default, one character per test plus listing at end
    - F.
    - . indicates success, F indicates failed, E indicates error
  - `verbosity=2`
    - `test_add (__main__.TestAdd) ... FAIL`
    - `test_add_op (__main__.TestAdd) ... ok`