Programming Principles in Python (CSCI 503/490)

OS Integration

Dr. David Koop
Debugging
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
  ```
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
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)
```
Python Debugger (pdb)

• Debuggers offer the ability to inspect and interact with code as it is running
  - Post-mortem inspection (%debug, python -m pdb)
  - Breakpoints (just call breakpoint())

• pdb is standard Python, also an ipdb variant for IPython/notebooks
  - 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
JupyterLab Debugger
JupyterLab Debugger
Python 3.11: Fine-Grained Error Locations

- Python 3.11 was released yesterday
- Code is faster (10-60% faster than 3.10, 25% average on benchmark)
- Debugging: Errors can show more specific locations
- Old Error:
  - Traceback (most recent call last):
    - File "distance.py", line 11, in <module>
      print(manhattan_distance(p1, p2))
    - File "distance.py", line 6, in manhattan_distance
      return abs(pt_1.x - pt_2.x) + abs(pt_1.y - pt_2.y)
    - AttributeError: 'NoneType' object has no attribute 'x'
Python 3.11: Fine-Grained Error Locations

- New Error:

  - Traceback (most recent call last):
    File "distance.py", line 11, in <module>
    print(manhattan_distance(p1, p2))
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    File "distance.py", line 6, in manhattan_distance
    return abs(pt_1.x - pt_2.x) + abs(pt_1.y - pt_2.y)
    ^^^^^^^^

    AttributeError: 'NoneType' object has no attribute 'x'
Python 3.11: Fine-Grained Error Locations

• Traceback (most recent call last):
  File "query.py", line 37, in <module>
    magic_arithmetic('foo')
  File "query.py", line 18, in magic_arithmetic
    return add_counts(x) / 25
      ^^^^^^^^^^^^^^-
  File "query.py", line 24, in add_counts
    return 25 + query_user(user1) + query_user(user2)
        ^^^^^^^^^^^^^^^^^^^^^-
  File "query.py", line 32, in query_user
    return count(db, response['a']['b']['c']['user'])
        ~~~~~~~~~~~~~~~~~~~~~^-^-
TypeError: 'NoneType' object is not subscriptable
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
- 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
Startup and Cleanup for Tests

- `setUp`: instantiate particular objects, read data, etc.
- `tearDown`: get rid of unnecessary objects
- Example: set up a GUI widget that will be tested
  
  ```python
  def setUp(self):
      self.widget = Widget(some_params)
  
  def tearDown(self):
      self.widget.dispose()
  ```

- Also functions for setting up classes and modules
Mock Testing

• Sometimes we don't want to actually execute all of the code that may be triggered by a particular test
• Examples: code that posts to Twitter, code that deletes files
• We can mock this behavior by substituting the actual methods with mockers
• Can even simulate side effects like having the function being mocked raise an exception signifying the network is done
Mock Examples

- Can check whether/how many times the mocked function was called

```python
from unittest.mock import MagicMock
thing = ProductionClass()
thing.method = MagicMock(return_value=3)
thing.method(3, 4, 5, key='value')
thing.method.assert_called_with(3, 4, 5, key='value')
```

- Can check whether/how many times the mocked function was called

```python
from unittest.mock import patch
with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
    thing = ProductionClass()
    thing.method(1, 2, 3)
mock_method.assert_called_once_with(1, 2, 3)
```
Assignment 6

- Object-Oriented Programming & Exceptions
- Classes for an online market
- Use Inheritance
- Due Nov. 1 (before the exam)
Test 2

- Thursday, November 3, 12:30-1:45pm in PM 253
- Covers material from the beginning of course, emphasizing material since Test 1
- Similar Format to Test 1
Integration with the Operating System

• For now, focus on the filesystem
  - Listing & Traversing Directories
  - Creating Directories
  - Matching Files
  - Copying, Moving, Removing Files/Directories

• Using Material by Vuyisile Ndlovu:
Modules

- In general, cross-platform! (Linux, Mac, Windows)
- os: translations of operating system commands
- shutil: better support for file and directory management
- fnmatch, glob: match filenames, paths
- os.path: path manipulations
- pathlib: object-oriented approach to path manipulations, also includes some support for matching paths
Directory Listing

• Old approach: `os.listdir`

• New approach: `os.scandir`
  - Uses iterators, object-based, faster (fewer stat calls), returns `DirEntry`
  - with `os.scandir('my_directory/')` as entries:
    ```python
    for entry in entries:
        print(entry.name)
    ```

• Pathlib approach:
  - from `pathlib` import `Path`
    ```python
    path = Path('my_directory/')
    for entry in path.iterdir():
        print(entry.name)
    ```
Listing Files in a Directory

- Difference between file and directory
- `isfile/is_file` methods:
  - `os.path.isfile`
  - `DirEntry.is_file`
  - `Path.is_file`
- Test while iterating through
  ```python
  from pathlib import Path
  basepath = Path('my_directory/')
  files_in_basepath = basepath.iterdir()
  for item in files_in_basepath:
    if item.is_file():
      print(item.name)
  ```
Listing Subdirectories

• Use `isdir/is_dir` instead

```python
from pathlib import Path

basepath = Path('my_directory/')
files_in_basepath = basepath.iterdir()
for item in files_in_basepath:
    if item.is_dir():
        print(item.name)
```
File Attributes

- Getting information about a file is "stat"-ing it (from the system call name)
- Names are similarly a bit esoteric, use documentation
- `os.stat` or use `.stat` methods on `DirEntry/Path`
- Modification time:
  ```python
  from pathlib import Path
  current_dir = Path('my_directory')
  for path in current_dir.iterdir():
    info = path.stat()
    print(info.st_mtime)
  ```
- Also can check existence: `path.exists()`
Making Directories

• Modify the filesystem
• Know where you **currently are** first
  - `os.getcwd()` or `Path.cwd()`: current working directory
• `os.mkdir`: single subdirectory
• `os.makedirs`: multiple subdirs
• `pathlib.Path.mkdir`: single or multiple directories (with `parents=True`)
• Can raise exceptions (e.g. file already exists)
• `from pathlib import Path
  p = Path('example_directory/')
  p.mkdir()`
Filename Pattern Matching

- `string.endswith/startswith`: no wildcards
- `fnmatch`: adds * and ? wildcards to use when matching (not just like regex!)
- `glob.glob`: treats filenames starting with . as special
  - can do recursive matchings (e.g. in subdirectories) using **
- `pathlib.Path.glob`: object-oriented version of glob
- `from pathlib import Path`
  ```python
  p = Path('.
  for name in p.glob('*.p*'):
    print(name)
  ```
Pathname Manipulation

• `os.path.split` returns tuple (dirname, basename)
  - can use `os.path.dirname/basename` to get these only
  - `os.path.split('/path/to/file.txt')` # ('/path/to', 'file.txt')

• `os.path.join`: inverse of split

• `os.path.splitext`: split filename and extension

• `pathlib.Path` has OOP versions:
  - `.parent/.name == dirname/basename`
  - `.stem/.suffix ~ splitext, also suffixes`
  - `/ operator (also joinpath ~ join)`
Traversing Directories and Processing Files

• os.walk

• for dirpath, dirnames, files in os.walk('.'):  
  print(f'Found directory: {dirpath}')
  for file_name in files:
    print(file_name)

• Returns three values on loop iteration:
  1. The name of the current directory
  2. A list of subdirectories in the current directory
  3. A list of files in the current directory

• topdown and followlinks arguments

• pathlib algorithms exist but DIY
Temporary Files and Directories

• tempfile knows system directories for storing temporary files
• deletes the file when it is closed
• from tempfile import TemporaryFile
  with TemporaryFile('w+t') as fp:
    fp.write('Hello universe!')
    fp.seek(0)
    fp.read()
    # File is now closed and removed
• Can also use in with statement (context manager)
• Can also create temporary directories
Deleting Files and Directories

- **Files:** `os.remove` or `os.unlink`, or `pathlib.Path.unlink`

- `from pathlib import Path`
  ```
  Path('home/data.txt').unlink()
  ```

- **Directories:** `rmdir` or `shutil.rmtree`
  - `rmdir` only works if the directory is **empty**
  - **Careful:** this deletes the entire directory (and everything inside it)
    - `shutil.rmtree('my_documents/bad_dir')`
Copying Files & Directories

- `shutil.copy`: copy file to specified directory
  ```python
  shutil.copy('path/to/file.txt', 'path/to/dest_dir')
  ```
- `shutil.copy2` preserves metadata, same syntax
- **Copy entire tree**: `shutil.copytree('data_1', 'data1_backup')`
Moving and Renaming Files/Directories

- Moving files or directories:
  - shutil.move('dir_1/', 'backup/')

- Renaming files or directories:
  - os.rename
  - pathlib.Path.rename
  - data_file = Path('data_01.txt')
    data_file.rename('data.txt')
Archives

- **zipfile**: module to deal with zip files
- **tarfile**: module to deal with tar files, can compress (tar.gz)
- Easier: `shutil.make_archive`
  - Specify base name, format, and root directory to archive
  - `shutil.make_archive('data/backup', 'tar', 'data/')`
- To extract, use `shutil.unpack_archive`