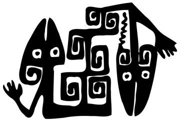
CS 3101-1 - Programming Languages: Python

Daniel Bauer (bauer@cs.columbia.edu)

October 15 2014



Contents

Programs are Error Prone

1. Syntax errors $\Leftarrow$ Detected by interpreter.
2. Errors at runtime $\Leftarrow$ Exception handling.
3. Incorrect program behavior (wrong result)

Programs are Error Prone

1. Syntax errors $\Leftarrow$ Detected by interpreter.
2. Errors at runtime $\Leftarrow$ Exception handling.
3. Incorrect program behavior (wrong result) $\Leftarrow$ **Debugging and testing**.

Debugging with print

- Most common way of debugging: Simply print intermediate results.
- Print all relevant information and reference to which part of the program prints.
- Better: Write debugging statements to `sys.stderr`.
- Comment/uncomment debugging code

```
def mirror(lst):
    for i in range(len(lst)):
        lst.append(lst[-i])
        #sys.stderr.write(
        #    "mirror: list for i={0}: ".format(i))
        #sys.stderr.write("{1}\n".format(lst))
    returnlst

x = [1,2,3]
print(mirror(x)) # Expected: [1,2,3,3,2,1]
```

logging module for debugging

- `logging` module to log errors and debugging messages.
- Provides central control over debugging output.

```
import logging
logging.basicConfig(level = logging.DEBUG)

def mirror(lst):
  for i in range(len(lst)):
      lst.append(lst[-i])
      logging.debug(
        "mirror: list for i={0}: {1} ".format(i, lst))
  return x
```

```
>>> mirror([1,2,3])
DEBUG:root:mirror: list for i=0: [1, 2, 3, 1]
...
```

logging - Logging levels

- Can output messages on different logging levels.

```
logging.basicConfig(level = logging.LEVEL)
```

level	function
logging.CRITICAL	logging.critical()
logging.ERROR	logging.error()
logging.WARNING	logging.warning()
logging.INFO	logging.info()
logging.DEBUG	logging.debug()

logging - more on logging

- Can output messages to a log file.

```
logging.basicConfig(level = logging.DEBUG,
                filename = 'bugs.log')
```

- Config is valid for all modules in a program.
  - Only set logfile and level once in main.
- Can add date and time:

```
logging.basicConfig(level = logging.DEBUG,
                filename = 'bugs.log',
                format='%(asctime)s %(message)s')
```

- More on logging:
  http://docs.python.org/library/logging.html

Python debugger - pdb

- Python provides a built-in debugger (module `pdb`).
- Allows to execute code line-by-line
- After each statement allows access to the process state.
- usage:
  - `import pdb`
  - insert `pdb.set_trace()` to indicate where to start debugging.
  - Just run the program from the command line.

Debugging - pdb commands

- ‘n’: next line.
- ‘p expression’: prints the value of the expression.
- ‘s’: step into subroutine.
- ‘c’: continue executing until next trace.

[Example in class.]

Using assert

- Allows you to insert simple tests in the code itself.
- `assert condition [, expression]`
- Interpreter evaluates condition.
  - if condition is `True`, nothing happens.
  - if conditions is `False`, an `AssertionError` is raised with `expression` as argument.
- Can also serve as documentation element.

```
def eat(self, item):
    assert isinstance(item, Food)
    self.stomach.append(item)
```

Testing module functionality with `main()`

- Modules that are not the main module, can use a `main` function for testing.
  - Only executed when file is run, not when it's imported.

```
def main():
    test = [1,2,3,4]
    assert my_reverse(test)== [4,3,2,1]
    print "Test passed."

if __name__ == "__main__":
    main()
```

  - Set up some input cases for testing (i.e. initialize test objects).
  - Process the test cases.
  - Check if the output is correct (using assert or just print).

[see tree.py example]

Unit Testing

- Need to test various units in a module (each function, method or class...), independently.
- Tests can fail for various reasons.
- More ‘permantent’ solution to check if code is still working as desired if parts are changed.
- Goes hand-in-hand with design specification.
- Python stdlib provides the `unittest` testing framework.

Defining Test Cases

- A test case is an elementary unit of tests (tests a single class, method, or function).
- Test cases are classes with base class `unittest.TestCase`.
- Methods: individual tests.
- Special method `setUp(self)` is run before every test to set up the test fixture (environment the test works in).
- Special method `tearDown(self)` performs cleanup after a test.

[see tree.py example]

Defining Test Cases - assert Methods

- instances of classes with base TestCase have assert methods, which will perform tests.

method	meaning
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

[see tree.py example]

Running tests

- Can invoke the test runner from within the module containing test cases:

```
if __name__ == '__main__':
    unittest.main()
```

  - Test all defined test cases.
- Can also run tests from the command line

```
$ python -m unittest tested_module
```

- Can run a specific test case

```
$ python -m unittest tested_module.TestCase
```

Automatic Documentation with pydoc and epydoc

- Automatically create documentation for modules and packages.
- Interpret code to indetify code structure.
- Include docstrings in documentation.
- Can use console interface to generate and browse documentation

```
$pydoc collections
```

- Can write HTML documentation:

```
$pydoc -w trees
$open trees.html
```

- Prettier HTML doc with epydoc `http://epydoc.sourceforge.net/`.

```
$epydoc epydoc-2.7 trees graphs dot_interface
$open html/index.html
```

Decorators

- Convenient concise way to modify classes, methods and functions.
- Are a form of meta-programming (like macros in C, annotations in Java...)
- Example uses:
  - Acquire and release some resource at entry/exit point.
  - Memoize previous computations performed by a function.
  - Log all errors in a function in a special way.
  - Make sure only a single instance of a class exists (singleton).
  - Make a method 'static'.
  - Make a method a 'class method'.

Decorators - Syntax

- Decorators are callable objects that are applied to functions or classes.

```
@dec2
@dec1
def func(arg1, arg2, ...):
    ...
```

is syntactic sugar for

```
def func(arg1, arg2, ...):
    ...
func = dec2(dec1(func))
```

Decorators can take arguments

- Can call a function to get a decorator.

```
@dec(foo1, foo2)
def func(arg1, arg2, ...):
    ...
```

is syntactic sugar for

```
def func(arg1, arg22, ...):
    ...
func = dec1(foo1, foo2)(func)
```

Example built-in decorators - `@staticmethod`

- `@staticmethod` creates a static method.
- Static methods do not receive implicit 'self' argument.
- Can be called on class and instance objects.

```
>>> class A():
...     @staticmethod
...     def foo(a1):
...         return a1 * 3
...
>>> A.foo(17)
51
>>> A().foo(17)
51
```

Writing Decorators - `@simplelog`

- Write a message to stderr everytime a method is entered and left.

```
>>> @simplelog
>>> def add(a,b):
...     return a + b
>>> x = 3
>>> add(2,x)
Calling add(2,3).
Result: 5
```

Writing Decorators - `@simplelog` implementation

```
def simplelog(fun):

  # New function object wraps fun
  def new_fun(*args):

      # Before fun is called
      argstr = ",".join([repr(arg) for arg in args])
      sys.stderr.write("Calling {0}({1}).\n".format(
                                  fun.__name__, argstr)

      result = fun(*args) # Call fun

      # After fun is called
      sys.stderr.write("Result: {0}\n".format(
                                   repr(result)))
      return result

  return new_fun
```

Writing Decorators - @memoized

```
>>> @memoized
... def factorial(n):
...     sys.stdout.write("#")
...     return 1 if n==1 else n * factorial(n-1)
...
>>> factorial(5)
#####120
>>> factorial(6)
#720
>>> factorial(4)
24
```

Writing Decorators - `@memoized` implementation

- Remember previous results of a computation.

```
def memoized(fun):
    cache = {}

    def new_fun(*args):
        argtuple = args
        if tuple(args) in cache:
            return cache[args]
        else:
            result = fun(*args)
            cache[args] = result
            return result

    return new_fun
```

Writing Decorators - `@singleton`

- Make sure that only a single instance of a class exists.
- Instance created the first time the class is instantiated.
- Instantiating the class again yields the same instance.

```
>>> @singleton
... class A(object):
...     pass
>>> x = A()
>>> y = A()
>>> x is y
True
```

Writing Decorators - `@singleton` implementation

- This is actually cheating (returns a function, not a class).
- Achieves desired behavior.
- Better solution requires more detail about Python internas.

```python
def singleton(cls):
  instances = {}

  def getinstance(*args, **kwargs):
    if cls not in instances:
        instances[cls] = cls(*args, **kwargs)
    return instances[cls]

  return getinstance
```

see `http://wiki.python.org/moin/PythonDecoratorLibrary` for more advanced decorators.

re - Regular Expressions

Regular Expressions

- Need to match and search complex patterns in text.
- Regular Expressions (RegEx) are used to describe sets of strings.
- Formally, can be used to define any regular language:
    - Languages recognized by finite state automata and left/right recursive grammars.

```
>>> import re
# RegEx to match Columbia UNIs
... uni_re = re.compile("[A-z]{2,3}[0-9]{4}")
>>> uni_re.search("My UNI is: xyz1234")
<_sre.SRE_Match object at 0x106da9e68>.
>>> uni_re.search("My UNI is xy.")
>>> uni_re.search("My UNI is 1234xy.")
>>> uni_re.search("My UNI is xyab1234.")
```

Regular Expressions - Basic Syntax

- Any literal character is a regular expression.
- if `R` and `S` are regular expressions:
  - `RS` is a regular expression (`R` followed by `S`).
  - `R|S` is a regular expression (either `R` or `S`).
- if `R` is a regular expression:
  - `(R)` is a regular expression.
  - R* is a regular expression (*Kleene star*, 0 or more repititions of `R`).

Example

`'a((ab)|(cd))*'` matches `'a'` followed by an arbitrary sequences of `'ab'` and `'cd'` blocks.

- `'a'`, `'aab'`, `'acd'`, `'aabcdabab'` etc.

Regular Expressions - Additional Repitition Pperators

- if `R` is a regular expression
  - `R?` is a regular expression (0 or 1 repititions of `R`)
  - `R+` is a regular expression (1 or more repititions of `R`)
  - `R{n}` is a regular expression (n repititions of `R`)
  - `R{n,m}` is a regular expression (at least n and at most m repititions of `R`)

Regular Expressions - Characer Groups and Ranges

- `.` matches any character except newline.
- `[abcd]` is shorthand for `a|b|c|d`.
- `[A-z]` matches any character between A and `z`. (`[A-Ba]` matches upper case letters or lower case a)
- `\s` matches any whitespace.
- `\w` matches any alphanumeric character and _
- `\d` matches any digit.

Note: Any special characters such as —*+?.\()[] need to be escaped with \ when used literaly.

Regular Expressions in Python - The re Module (1)

- import re module (near the beginning of the .py file).

```
import re
```

- match(re, string) checks if re matches a prefix of string.
- search(re, string) searches the string for an occurence of re.
- Both return a *match object* if a match is found, otherwise None.

```
>>> re.match("ab*", "abbbcd")
<_sre.SRE_Match object at 0x106da9e68>
>>> re.match("ab*", "cdabbbc")
>>> re.search("ab*", "cdabbbc")
<_sre.SRE_Match object at 0x1091a6cf0>
```

Regular Expressions in Python - The re Module (2)

- finditer(re, string) returns an iterator over all on-overlapping matches.
- Can pre-compile the regular expression into a pattern object for efficiency.

```
>>> matcher = re.compile("ab*")
>>> type(matcher)
<type '_sre.SRE_Pattern'>
>>> list(matcher.finditer("cdabbabc"))
[<_sre.SRE_Match object at 0x1096dbd98>, <_sre.
    SRE_Match object at 0x1096dbe00>]
```

Regular Expressions - Match Objects

- Match objects contains:
  - positional information about the match in the string:
    - `match.start()` start index of the match.
    - `match.end()` end index of the match.

```
>>> s = "cabbbcd"
>>> match = re.search("ab*", s)
>>> match.start()
1
>>> match.end()
5
>>> s[match.start(): match.end()]
'abbb'
```

Regular Expressions - Groups

- Match object contains copies of subsequences of the string
  - Corresponding to () groups in the regular expression.
  - Groups are indexed outside-in and left-to-right.
- Can name groups with `?P<name>`.

```
>>> match = re.search("(?P<agroup>a)((b*)c)","cabbbc")
>>> match.group(0) # Entire match
'abbbc'
>>> match.group(1)
'a'
>>> match.group(2)
'bbbc'
>>> match.group(3)
'bbb'
>>> match.group("agroup")
'a'
```

Regular Expressions - Groups

- Match object contains copies of subsequences of the string
  - Corresponding to () groups in the regular expression.
  - Groups are indexed outside-in and left-to-right.
- Can name groups with ?P<name>.

```
>>> match = re.search("(?P<agroup>a)((b*)c)","cabbbc")
>>> match.group(0) # Entire match
'abbbc'
>>> match.group(1)
'a'
>>> match.group(2)
'bbbc'
>>> match.group(3)
'bbb'
>>> match.group("agroup")
'a'
```

More about regExs:
http://docs.python.org/howto/regex.html.

The Zen of Python (1)

When in doubt...

```
>>> import this
```

The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.

The Zen of Python (2)

In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!