Inheritance Continued & Exceptions

David E. Culler
CS8 – Computational Structures in Data Science
http://inst.eecs.berkeley.edu/~cs88

Lecture 10
October 29, 2018

Notebooks from L09+L10: http://bit.ly/cs88-fa18-L09

Computational Concepts Toolbox

- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
- Dictionaries
- Data structures
- Tuple assignment
- Function Definition Statement
- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.
- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions
- Recursion
  - Linear, Tail, Tree
- Abstract Data Types
- Mutation
- Class
  - Object Oriented Programming
  - Inheritance

Administrative Issues

- Project 2 "Wheel" goes out this week
  - Discussion in lab
- There will be no Project 3
- Reading: (2.5-7), 2.9 , exceptions: 3.3

Today:

- Review Class concept
- Using class to create and manipulate objects
- Inheritance to specialize a class
  - Create subtypes of the object type
- Exceptions
  - Unprogrammed control transfers to catch unusual situations or errors
  - How they arise
  - How to handle exception
  - How to raise your own

Review: Python class

```
class <ClassName>:

    def <method-1>(self, ..)
      self.<instance_attr> = …
    .
    .
    .
    def <method-N>
```

https://docs.python.org/3/tutorial/classes.html

Class names should normally use the **CapWords** convention.

https://www.python.org/dev/peps/pep-0008/

Creating an object, invoking a method

The Class Constructor

```
my_acct = Account("David Culler", 93)
my_acct.withdraw(42)
```

Review: class example

```
class Account:
    # Class astributes outside and class defs
    _account_number_seed = 1000

    # Constructor
    def __init__(self, name, initial_deposit):
        # Initialize the instance attributes
        self._name = name
        self._acct_no = Account._account_number_seed
        Account._account_number_seed += 1
        self._balance = initial_deposit
        # Return None

    # Selectors
    def account_name(self):
        return self._name
    . . .
    def account_number(self):
        return self._acct_no
    . . .
```

class attributes

The object

private instance attributes, dot notation

class attributes, dot notation

object namespace

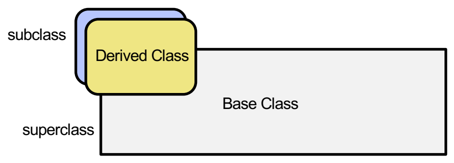
Methods

Inheritance

- Define a class as a specialization of an existing class
- Inherent its attributes, methods (behaviors)
- Add additional ones
- Redefine (specialize) existing ones
  - Ones in superclass still accessible in its namespace

```
class ClassName ( <inherits> ):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Inheritance



Example

```
class CheckingAccount(Account):

    def __init__(self, name, initial_deposit):
        # Use superclass initializer
        Account.__init__(self, name, initial_deposit)
        # Additional initialization
        self._type = "Checking"

    def account_type(self):
        return self._type

    # Display representation
    def __repr__(self):
        return '<' + str(self.account_type()) + 'Account:…'
```

Attribute in subclass, not in superclass

Another Example

```
class SavingsAccount(Account):

    interest_rate = 0.02

    def __init__(self, name, initial_deposit):
        # Use superclass initializer
        Account.__init__(self, name, initial_deposit)
        # Additional initialization
        self._type = "Savings"

    def account_type(self):
        return self._type

    def acrue_interest(self):
        self._balance = self._balance *
                         (1 + SavingsAccount.interest_rate)
```

Methods in subclass, not in superclass

Classes using classes

```
class Bank:
    _accounts = []

    def add_account(self, name, account_type, initial_deposit):
        if account_type == 'Savings':
            new_account = SavingsAccount(name, initial_deposit)
        elif account_type ==  'Checking':
            new_account = CheckingAccount(name, initial_deposit)
        else:
            assert True, "Bad Account type: " + account_type
        assert initial_deposit > 0, "Bad deposit"

        Bank._accounts.append(new_account)
        return new_account

    def accounts(self):
        return self._accounts[:]

    def show_accounts(self):
        for acct in self.accounts():
            print(acct.account_number(), acct.account_type(),
                  acct.account_name(), acct.account_balance())
```

Key concepts to take forward

- Classes embody and allow enforcement of ADT methodology
- Class definition
- Class namespace
- Methods
- Instance attributes (fields)
- Class attributes
- Inheritance
- Superclass reference

Additional examples

- Redesign our KV as a class
- How should “new KV” vs mutation be handled
- Inheritance and “new object” in superclass

KV as a true object

```
class KV:
    """Key-Value container abstractionma collection of key-value pairs"""
    def __init__(self, kv_pairs=[]):
        self._kv = []
        for (key, val) in kv_pairs:   # Verify and initialize
            assert (type(key) == str) # the key should be a string
            self._kv.append((key, val))

    def items(self):
        """Return a list of the (key, value) pairs in kv."""
        return self._kv

    def get(self, key):
        """Return the value bound to key in kv, or None if not present."""
        for k, v in self.items():
            if k == key:
                return v
        return None

    def keys(self):
        """Return a list of the keys in kv"""
        return [key for (key, val) in self.items()]

    def values(self):
        """Return a list of the values in kv"""
        return [val for (key, val) in self.items()]

    def add(self, key, value):
        """Return a new KV adding binding (key, value)"""
        return KV([(key, value)] + self.items())

    def delete(self, key):
        """Return a new KV having removed any binding for key"""
        return KV([(k, v) for (k, v) in self.items(kv) if not k == key])
```

Class methods

- Defined on the class
  - rather than objects of the class
  - Like class attributes
- Indicated by `@classmethod`
  - Take a class argument, rather than self

```
class KV:
    """Key-Value container abstraction
    a collection of key-value pairs such that kv_get(kv, key) returns the
value
    """
    def __init__(self, kv_pairs=[]):
        self._kv = []
        for (key, val) in kv_pairs:   # Verify and initialize
            assert (type(key) == str) # the key should be a string
            self._kv.append((key, val))

    @classmethod
    def create(cls, kv_pairs=[]):
        return cls(kv_pairs)
```

Inheritance Example

```
class KVnodup(KV):
    def __init__(self, kv_pairs=[]):
        self._kv = []
        for (key, val) in kv_pairs:   # Verify that initialization is vali
            assert type(key) == str   # the key should be a string
            if not key in self:
                self._kv.append((key, val))
```

Subclass type

Explicit use of class constructor – interferes with inheritance

```
def add(self, key, value):
    """Return a new KV adding binding (key, value)"""
    return KV([(key, value)] + self.items())
```

Use type(self) as constructor to maintain inherited type

```
def add(self, key, value):
    """Return a new KV adding binding (key, value)"""
    return type(self)([(key, value)] + self.items())
```

Exception (read 3.3)

- Mechanism in a programming language to declare and respond to “exceptional conditions”
  - enable non-local cntinuations of control
- Often used to handle error conditions
  - Unhandled exceptions will cause python to halt and print a stack trace
  - You already saw a non-error exception – end of iterator
- Exceptions can be handled by the program instead
  - `assert, try, except, raise` statements
- Exceptions are objects!
  - They have classes with constructors

Handling Errors

- Function receives arguments of improper type?
- Resource, e.g., file, is not available
- Network connection is lost or times out?

1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.
1545 Relay #70 Panel F (moth) in relay.
First actual case of bug being found.
1630 antangent started.
1700 closed down.

Grace Hopper's Notebook, 1947, Moth found in a Mark II Computer

Example exceptions

notebook

```
>>> 3/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> str.lower(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor 'lower' requires a 'str' object
but received a 'int'
>>> ""[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

- Unhandled, thrown back to the top level interpreter
- Or halt the python program

Functions

- Q: What is a function supposed to do?
- A: One thing well
- Q: What should it do when it is passed arguments that don't make sense?

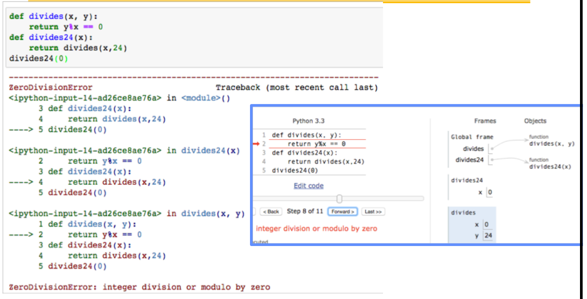
```
>>> def divides(x, y):
...     return y%x == 0
...
>>> divides(0, 5)
???

>>> def get(data, selector):
...     return data[selector]
...
>>> get({'a': 34, 'cat':'9 lives'}, 'dog')

????
```

Exceptional exit from functions

```
>>> def divides(x, y):
...     return y%x == 0
...
>>> divides(0, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in divides
ZeroDivisionError: integer division or modulo by zero
>>> def get(data, selector):
...     return data[selector]
...
>>> get({'a': 34, 'cat':'9 lives'}, 'dog')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in get
KeyError: 'dog'
>>>
```

- Function doesn't “return” but instead execution is thrown out of the function

Continue out of multiple calls deep



```
def divides(x, y):
    return y%x == 0
def divides24(x):
    return divides(x,24)
divides24(0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-14-ad26ce8ae76a> in <module>()
      3 def divides24(x):
      4     return divides(x,24)
----> 5 divides24(0)

<ipython-input-14-ad26ce8ae76a> in divides24(x)
      2     return y%x == 0
      3 def divides24(x):
----> 4     return divides(x,24)
      5 divides24(0)

<ipython-input-14-ad26ce8ae76a> in divides(x, y)
      1 def divides(x, y):
----> 2     return y%x == 0
      3 def divides24(x):
      4     return divides(x,24)
      5 divides24(0)

ZeroDivisionError: integer division or modulo by zero
```

Python 3.3

```
def divides(x, y):
    return y%x == 0
def divides24(x):
    return divides(x,24)
divides24(0)
```

integer division or modulo by zero

- Stack unwinds until exception is handled or top

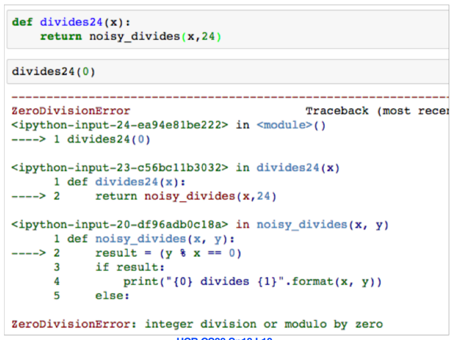
Types of exceptions

- `TypeError` -- A function was passed the wrong number/type of argument
- `NameError` -- A name wasn't found
- `KeyError` -- A key wasn't found in a dictionary
- `RuntimeError` -- Catch-all for troubles during interpretation
- . . .

Demo

Flow of control stops at the exception

- And is 'thrown back' to wherever it is caught



```
def divides24(x):
    return noisy_divides(x,24)
```

```
divides24(0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recen
<ipython-input-24-ea94e81be222> in <module>()
----> 1 divides24(0)

<ipython-input-23-c56bc11b3032> in divides24(x)
      1 def divides24(x):
----> 2     return noisy_divides(x,24)

<ipython-input-20-df96adb0c18a> in noisy_divides(x, y)
      1 def noisy_divides(x, y):
----> 2     result = (y % x == 0)
      3     if result:
      4         print("{0} divides {1}".format(x, y))
      5     else:

ZeroDivisionError: integer division or modulo by zero
```

Assert Statements

- Allow you to make assertions about assumptions that your code relies on
  - Use them liberally!
  - Incoming data is dirty till you've washed it

```
assert <assertion expression>, <string for failed>
```

- Raise an exception of type `AssertionError`
- Ignored in optimize flag: python3 –O …
  - Governed by bool __debug__

```
def divides(x, y):
    assert x != 0, "Denominator must be non-zero"
    return y%x == 0
```

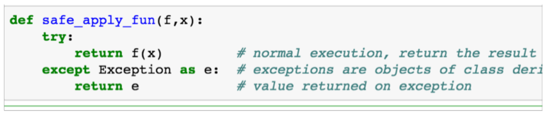
Handling Errors – `try` / `except`

- Wrap your code in `try` – `except` statements

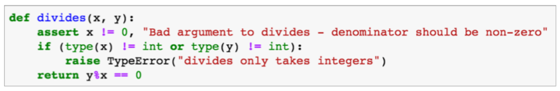
```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
... # continue here if <try suite> succeeds w/o exception
```

- Execution rule
  - <try suite> is executed first
  - If during this an exception is raised and not handled otherwise
  - And if the exception inherits from <exception class>
  - Then <except suite> is executed with <name> bound to the exception
- Control jumps to the except suite of the most recent `try` that handles the exception

Demo



```
def safe_apply_fun(f,x):
    try:
        return f(x)            # normal execution, return the result
    except Exception as e:     # exceptions are objects of class deri
        return e               # value returned on exception
```



```
def divides(x, y):
    assert x != 0, "Bad argument to divides - denominator should be non-zero"
    if (type(x) != int or type(y) != int):
        raise TypeError("divides only takes integers")
    return y%x == 0
```

Raise statement

- Exception are raised with a `raise` statement\

```
raise <exception>
```

- <expression> must evaluate to a subclass of BaseException or an instance of one
- Exceptions are constructed like any other object

```
TypeError('Bad argument')
```

Exceptions are Classes

```
class NoiseyException(Exception):
    def __init__(self, stuff):
        print("Bad stuff happened", stuff)
```

```
try:
    return fun(x)
except:
    raise NoiseyException((fun, x))
```

Demo

Summary

- Approach creation of a class as a design problem
  - Meaningful behavior => methods [& attributes]
  - ADT methodology
  - What's private and hidden? vs What's public?
- Design for inheritance
  - Clean general case as foundation for specialized subclasses
- Use it to streamline development
- Anticipate exceptional cases and unforeseen problems
  - try … catch
  - raise / assert

Computational Concepts Toolbox

- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
- Dictionaries
- Data structures
- Tuple assignment
- Function Definition Statement
- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.

- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions
- Recursion
- Abstract Data Types
- Mutation
- Class
  - Object Oriented Programming
  - Inheritance
- Exceptions