OOP Recap

Miscellaneous topics to finish off the unit on OOP
  • histogram (example of inheritance)
  • unit testing (based on OOP concepts)
  • a few notes on encapsulation

Histogram Class

A histogram (aka “bar chart”) is a collection of counts

```python
>>> vowels = list('aeiou')
>>> h = Histogram(vowels)
>>> for ch in 'the quick brown fox jumped over the lazy dog':
...   if ch in vowels:
...     h.count(ch)
```

![Histogram Class Demo](image)
We could use a Python dictionary, but
  • it would be nice if counts are initialized to 0
  • don't allow random updates (e.g. h[‘a’] += 3)

Instead: define new class that is derived from Python’s dictionary class

class Histogram(dict):

Examples using the new class:

```python
>>> vowels = list('aeiou')

>>> h = Histogram(vowels)
>>> h
{'u': 0, 'e': 0, 'o': 0, 'i': 0, 'a': 0}

>>> h.count('e')

>>> h
{'u': 0, 'e': 1, 'o': 0, 'i': 0, 'a': 0}

>>> h['e'] += 3
Exception: can't assign to histogram; use count to update a bin

>>> list(h.values())
[0, 1, 0, 0, 0]

>>> list(h.keys())
['u', 'e', 'o', 'i', 'a']
```
See definition in Histogram.py

Things to look for:

(1) The constructor calls the parent class constructor

```python
def __init__(self, arg):
    dict.init( self, ... )
```

The previous example (Queue) did not have a constructor; new Queues are initialized just like new lists

In this case, we're passing some values to the dictionary constructor so it initializes the values (all 0's)
(2) The constructor validates the argument it is passed

We're expecting either a list or tuple of strings, which will be the labels on the “bins” in the histogram

```python
h = Histogram(['green', 'yellow', 'white', 'steel', '?'])
h = Histogram(('alpha', 'beta', 'gamma', 'delta', ...))
```

A builtin function named `isinstance` returns True if the argument belongs to a specified class (or tuple of classes)

Why use `isinstance`?

• clarity:

```python
if isinstance(arg, (list, tuple))
is easier to follow than
if type(arg) == list or type(arg) == tuple
```

• knows about inheritance hierarchy:

```python
>>> h = Histogram(vowels)
>>> type(h) == dict
False
>>> isinstance(h, dict)
True
```

**aside: minimize type checking**

Good OOP style allows the system to figure out types, “bind” function calls at runtime (example: points function for cards)

Don’t try to over-think and add type checks

One place they are useful, though, is constructors: raise an error when an object is created, not later, when it is used
The constructor uses dictionary comprehension

Like list comprehension, except it creates a dictionary, not a list

```python
>>> a = [ n**2 for n in range(10) ]
>>> d = { n : n**2 for n in range(10) }
```

Hint: you might want to use a similar construct in your Deck class...

A call to `list(x)` creates a new list object, initialized using the items in `x`

Similarly, `dict(x)` initializes a dictionary using elements in `x`

```
class Foo(dict):
    def __init__(self, a):
        for x in a:
            self[x] = 0
```

OK:

```
class Foo(dict):
    def __init__(self, a):
        for x in a:
            self[x] = 0
```

Better:

```
class Foo(dict):
    def __init__(self, a):
        dict.__init__(self, { x:0 for x in a })
```

Why is the second version better?

General rule: if you write `__init__` for a new class, have it call the parent class `__init__` function, then add your own attributes
Unit Tests: Another Example of OOP

Python’s unit test framework is based on similar modules in other languages

JUnit, EUnit (Erlang), HUnit (Haskell), ...
(see List of Unit Testing Frameworks on Wikipedia (!))

Code at the beginning of test_Cards.py:

```python
import unittest
from Card import *
class CardTest(unittest.TestCase):

Inside the CardTest class:

def test_01_cards(self):
  ...
def test_02_points(self):
  ...
def test_03_blackjack_cards(self):
  ...
```

When the program is run from the command line, the module searches for all methods with names that start test_ and calls them

Benefits:
• all tests are run, even if earlier tests raise exceptions
• convenient methods for testing expected values, printing messages when tests fail
• summary statistics about number of tests

Much more — read on your own