EECS 390 – Lecture 11

Functional Data Abstraction

3/3/24

Data Abstraction

- Abstraction separates what something is from how it works
- ***Abstract data types (ADTs)*** separate the interface of a data type from its implementation
- ***Encapsulation*** is an important, though not universal, property of an ADT, binding the data the ADT represents along with the functions that operate on that data
- We will build a hierarchy of ADTs, beginning with immutable pairs all the way up to an abstraction similar to that provided by object-oriented programming

Pair ADT

- Recall that nested functions allow us to store data in the non-local environment:

```
def make_greater_than(threshold):
    def greater_than(x):
        return x > threshold
    return greater_than
```

- Let's use this to define a pair abstraction:

```
def pair(x, y):
    def get(i):
        return x if i == 0 else y
    return get

def first(p):
    return p(0)

def second(p):
    return p(1)
```

```
>>> p = pair(3, 4)
>>> first(p)
3
>>> second(p)
4
```

Mutable Pair ADT

- The `pair` ADT is immutable, but mutation is important in imperative programming
- Mutable pair:

```
def mutable_pair(x, y):
    def get(i):
        return x if i == 0 else y

    def set(i, value):
        nonlocal x, y
        if i == 0:
            x = value
        else:
            y = value

    return pair(get, set)
```

Separate get and set functions

Use an immutable pair to return the two functions

Using a Mutable Pair

- Accessor functions:

```
def mutable_first(p):
    return first(p)(0)

def mutable_second(p):
    return first(p)(1)

def set_first(p, value):
    second(p)(0, value)

def set_second(p, value):
    second(p)(1, value)
```

Need to avoid name clash between immutable and mutable pair functions

```
>>> p = mutable_pair(3, 4)
>>> mutable_first(p)
3
>>> mutable_second(p)
4
>>> set_first(p, 5)
>>> set_second(p, 6)
>>> mutable_first(p)
5
>>> mutable_second(p)
6
```

Message Passing

- Rather than defining external functions for each behavior, we can use pass a ***message*** requesting a particular behavior
- A ***dispatch function*** takes the appropriate action given a message

```
>>> p = mutable_pair(3, 4)
>>> p('first')
3
>>> p('second')
4
>>> p('set_first', 5)
>>> p('set_second', 6)
>>> p('first')
5
>>> p('second')
6
```

```
def mutable_pair(x, y):
    def dispatch(message, value=None):
        nonlocal x, y
        if message == 'first':
            return x
        if message == 'second':
            return y
        if message == 'set_first':
            x = value
        elif message == 'set_second':
            y = value
    return dispatch
```

List ADT

- We can use pairs to implement a recursive list

```
def mutable_list():
    empty_list = None
    head = empty_list
    tail = empty_list
    ...
    def dispatch(message, arg1=None, arg2=None):
        if message == 'len':
            return size(head)
        if message == 'getitem':
            return getitem(head, arg1)
        if message == 'setitem':
            return setitem(head, arg1, arg2)
        if message == 'str':
            return to_string()
        if message == 'append':
            return append(arg1)
    return dispatch
```

Representation of empty list

Keep track of both ends

List Functions

- Locally defined functions for `len`, `getitem`, `setitem`:

```
def size(mlist):
    if mlist is empty_list:
        return 0
    return 1 + size(mlist('second'))

def getitem(mlist, i):
    if i == 0:
        return mlist('first')
    return getitem(mlist('second'), i - 1)

def setitem(mlist, i, value):
    if i == 0:
        mlist('set_first', value)
    else:
        setitem(mlist('second'), i - 1, value)
```

List Functions

- Locally defined functions for `append`, `str`:

```
def append(value):
    nonlocal head, tail
    if head is empty_list:
       head = mutable_pair(value, empty_list)
       tail = head
    else:
       tail('set_second',
            mutable_pair(value, empty_list))
       tail = tail('second')

def to_string():
    if head is empty_list:
       return '[]'
    return ('[' + str(head('first')) +
            to_string_helper(head('second')) + ']')

def to_string_helper(mlist):
    if mlist is empty_list:
        return ''
    return (', ' + str(mlist('first')) +
            to_string_helper(mlist('second')))
```

Using a List

- We use a list by passing it a message and the required arguments

```
>>> l = mutable_list()
>>> l('str')
'[]'
>>> l('len')
0
>>> l('append', 3)
>>> l('append', 4)
>>> l('append', 5)
>>> l('str')
'[3, 4, 5]'
>>> l('len')
3
>>> l('getitem', 1)
4
>>> l('setitem', 1, 6)
>>> l('str')
'[3, 6, 5]'
```

Dictionary ADT

- Now that we have lists, we can use them to implement a dictionary as a list of key-value pairs

Helper function to find key-value pair

```
def dictionary():
    records = mutable_list()
    ...
    def get_record(key):
        size = records('len')
        i = 0
        while i < size:
            record = records('getitem', i)
            if key == record('first'):
                return record
            i += 1
        return None

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        if message == 'setitem':
            setitem(key, value)

    return dispatch
```

Dictionary Functions

- Locally defined functions for `getitem`, `setitem`

```
def getitem(key):
    record = get_record(key)
    return (record('second') if
            record is not None else None)

def setitem(key, value):
    record = get_record(key)
    if record is None:
        records('append',
                mutable_pair(key, value))
    else:
        record('set_second', value)
```

Using a Dictionary

- Comparison of our dictionary with built-in Python `dict`:

```
>>> d = dictionary()
>>> d('setitem', 'a', 3)
>>> d('setitem', 'b', 4)
>>> d('getitem', 'a')
3
>>> d('getitem', 'b')
4
>>> d('setitem', 'a', 5)
>>> d('getitem', 'a')
5
```

```
>>> d = dict()
>>> d.__setitem__('a', 3)
>>> d.__setitem__('b', 4)
>>> d.__getitem__('a')
3
>>> d.__getitem__('b')
4
>>> d.__setitem__('a', 5)
>>> d.__getitem__('a')
5
```

Dispatch Dictionaries

- Now that we have dictionaries, we can use them to simplify the dispatch behavior of our ADTs
- ***Dispatch function***: lengthy conditional that compares the input message to each known message and performs the appropriate action
  - In our implementation, must take in maximum number of arguments for any behavior
- ***Dispatch dictionary***: store mapping of message to function that performs that behavior in a dictionary
  - Dispatch function now just looks up message in dictionary and returns the corresponding function

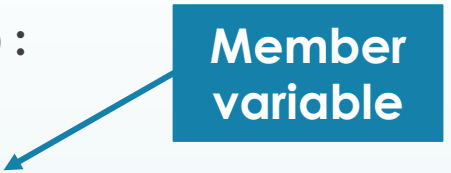
Bank Account ADT

- Implementation using a dispatch dictionary:

```
def account(initial_balance):
    ...
    dispatch = dictionary()
    dispatch('setitem', 'balance', initial_balance)
    dispatch('setitem', 'deposit', deposit)
    dispatch('setitem', 'withdraw', withdraw)
    dispatch('setitem', 'get_balance', get_balance)

    def dispatch_message(message):
        return dispatch('getitem', message)

    return dispatch_message
```



Bank Account Functions

- Locally defined functions for deposit, withdraw, get_balance:

```
def deposit(amount):
    new_balance = (dispatch('getitem',
                            'balance') + amount)
    dispatch('setitem', 'balance', new_balance)
    return new_balance

def withdraw(amount):
    balance = dispatch('getitem', 'balance')
    if amount > balance:
        return 'Insufficient funds'
    balance -= amount
    dispatch('setitem', 'balance', balance)
    return balance

def get_balance():
    return dispatch('getitem', 'balance')
```

Using a Bank Account

- Comparison of our bank account ADT with one implemented using a Python class:

```
>>> a = account(33)
>>> a('get_balance')()
33
>>> a('deposit')(4)
37
>>> a('withdraw')(7)
30
>>> a('withdraw')(77)
'Insufficient funds'
```

```
>>> a = account(33)
>>> a.get_balance()
33
>>> a.deposit(4)
37
>>> a.withdraw(7)
30
>>> a.withdraw(77)
'Insufficient funds'
```