Principles of Programming Languages

Lesson # 7

Object-Oriented Programming

Do you remember???

- What is it message passing?
- What is it dispatch function?
- What is it dispatch dictionary?
- Which types did we implement last lesson?
- Are they mutable or immutable?
- What does enable us to implement mutable data?
- What are restrictions of a dictionary?
- What is it hashable type?

Intro: functional implementation => dispatch dictionary

- Functional implementation = *dispatch function* that gets messages as arguments and performs operations on local state variables

- Example: **make_account()**

```python
def make_account(balance, owner):
    """Return a dispatch function that represents a bank account."""
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    def deposit(amount):
        nonlocal balance
        balance = balance + amount
        return balance
    def get_balance():
        return balance
    def get_owner():
        return owner
    def dispatch(msg):
        if msg == 'withdraw':
            return withdraw
        elif msg == 'deposit':
            return deposit
        elif msg == 'get_balance':
            return get_balance
        elif msg == 'get_owner':
            return get_owner
    return dispatch
```

Manipulating account

```
>>> a = make_account(100, ‘M’)
>>> a(‘get_owner’)()
'M'
>>> a('get_balance')()
100
>>> a('withdraw‘)(20)
80
```

Intro: functional implementation => dispatch dictionary

- Add more queries: set_balance, set_owner,…
  - elif for each message/operation

- Better organization – store <name, operation> pairs in a dictionary
  - Improved implementation = ***dispatch dictionary***
  - Manipulating = retrieving operations by their names

- Example: make_account()

```python
def make_account(balance, owner):
    """Return a dispatch function that represents a bank account."""
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    def deposit(amount):
        nonlocal balance
        balance = balance + amount
        return balance
    def get_balance():
        return balance
    def get_owner():
        return owner
    dispatch = {'withdraw': withdraw, 'deposit': deposit,
                'get_balance': get_balance, 'get_owner': get_owner}
    return dispatch
```

Manipulating account

Dispatch function

```
>>> a = make_account(100, "M")
>>> a("get_owner")()
'M'
>>> a('get_balance')()
100
>>> a('withdraw‘)(20)
80
```

Dispatch dictionary

```
>>> a = make_account(100, "M")
>>> a["get_owner"]()
'M'
>>> a['get_balance']()
100
>>> a['withdraw'](20)
80
```

OOP

- A method for organizing programs
- Like *abstract data types*, create an abstraction barrier
- Like *dispatch dictionaries* in message passing, respond to behavioral requests
- Like *mutable data structures*, objects have local state

Object system

- An object is a data value that has *methods* and *attributes*, accessible via *dot notation*

```
>>> d = date(2022,12,13)
>>> d.day
13
>>> d.strftime('%A, %B %d')
'Tuesday, December 13'
```

- Every object also has a type, called a class

```
>>> type(d)
<class 'datetime.date'>
```

Objects and Classes

- A ***class*** serves as a *template* for all objects whose type is that class

- Every ***object*** is an *instance* of a particular class

- New classes can be defined similarly to how new functions can be defined

מה לגבי הערכים של שדות ומתודות? האם הם משותפים?

- A class definition specifies the <u>attributes</u> and <u>methods</u> *shared among objects* of that class

Example: bank account

- Bank accounts are naturally modeled as mutable values that have a balance.

- Account’s behavior:
  – make a ***withdraw***
  – return its current ***balance***,
  – return the name of the account ***holder***, and
  – accept ***deposits***.

Creating a new object

- An **Account** class allows us to create multiple instances of bank accounts
  - creating a new object instance – ***instantiating*** *the class*

- The syntax = syntax of calling a function

- Example - creating account of Jim:

```
>>> a = Account('Jim')
```

Attributes

- An attribute of an object is a *name-value* pair associated with the object, which is accessible via dot notation.

- The attributes specific to a particular object are called **instance attributes**.
  - *balance* and account *holder name*

- May also be called **fields**, **properties**, or **instance variables**.

```
>>> a.holder
'Jim'
>>> a.balance
0
```

Methods

- **Methods** - functions that operate on the object or perform object-specific computations

- The side effects and return value of a method can depend upon *other attributes of the object (and change them)*

- Example – ***deposit***:
  - <u>*takes*</u> one argument (*amount*),
  - <u>*changes*</u> the *balance* attribute, and
  - <u>*returns*</u> the resulting *balance*

```
>>> a.deposit(15)
15
```

Invoking methods

- In OOP, we say that methods are **invoked on a particular object**.
- Example: result of invoking the *withdraw* method either
  - the withdrawal is approved and the balance is deducted and returned, or
  - the request is declined and the account prints an error message.

```
>>> a.withdraw(10) # The withdraw method returns the balance after withdrawal
5
>>> a.balance # The balance attribute has changed
5
>>> a.withdraw(10)
'Insufficient funds'
```

Defining Classes

- Class statements consist of a single clause
  - Defines the class name and a base class
  - Includes a suite of statements to define the attributes of the class:

```
class <name>(<base class>):
   <suite>
```

class statement execution

- A new class is created and bound to <name> in the first frame of the current environment

- The suite is then executed
  - Any names bound within the <suite> of a class statement (through def or assignment statements) *create* or *modify attributes* of the class in the local (class's or object's) frame

Initializing

- Classes are organized around manipulating instance attributes
  - the name-value pairs associated with each *object* of that class
- The class specifies the *instance attributes* of its objects by defining a method for ***initializing*** new objects.
- Example: initializing an object of Account - assigning its starting balance to 0.

Instance attribute

Constructor

- The <suite> of a class statement contains ***def*** statements that define *new methods* for objects of that class.

- **Constructor** - the method that initializes objects
  – has a special name in Python, **__init__**

Example

```
>>> class Account(object):
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
```

__init__ method has two formal parameters:

1. **self** is bound to the newly created Account object
2. **account_holder** is bound to the argument passed to the class when it is called to be instantiated

Example

- The constructor binds the instance attributes:
  - balance to *0,*
  - holder to the value of the *account_holder*.

- account_holder is *local* to the __init__
- holder is stored as an attribute of self and *persists*.

Example: Instantiating

Having defined the Account class, we can instantiate it:

>>> a = Account('Jim')

1. creates a new object that is an instance of Account,
2. calls the constructor function __init__ with two arguments: the newly created object and the string 'Jim'.

Convention: the parameter name ***self*** is the **first** argument of a constructor.

Can be of ANY other name

Example: Accessing attributes

We can access the object’s balance and holder using dot notation:

```
>>> a.balance
0
>>> a.holder
'Jim'
```

Example: Identity

Each *new* account instance has its own balance attribute, the value of which is independent of other objects of the same class

```
>>> b = Account('Jack')
>>> b.balance = 200
>>> [acc.balance for acc in (a, b)]
[0, 200]
```

Identity

- Every object that is an instance of a user-defined class has a unique **identity**
- Object identity is compared by the **is** and **is not** operators:

>>> a **is** a

True

>>> a **is not** b

True

Sharing

- Binding an object to a new name using assignment does not create a new object

```
>>> c = a
>>> c is a
True
```

- New objects of user-defined classes are only created when a class is instantiated

Methods

- **Methods** are defined by a ***def statement*** in the suite of a class

```
>>> class Account(object):
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        def deposit(self, amount):
            self.balance = self.balance + amount
            return self.balance
        def withdraw(self, amount):
            if amount > self.balance:
                return ’Insufficient funds’
            self.balance = self.balance - amount
            return self.balance
```

deposit and **withdraw** are both defined as methods on objects of the Account class

Methods vs. Functions

- The function value that is created by a def statement within a class statement is bound to the declared name **locally** within the class as an attribute.

- That value is invoked as a method using dot notation from an instance of the class.

- Each method definition has a special first parameter **self**, which is bound to the object on which the method is invoked.

Methods vs. Functions

- Example:
  - deposit is invoked on a particular Account object and passed a single argument value: *amount*
  - The object itself is bound to ***self***, and the argument is bound to *amount*.

- All invoked methods have access to the object via the **self** parameter to access and manipulate the object's state.

Invoke methods

Use dot notation:

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
>>> tom_account.withdraw(90)
10
>>> tom_account.withdraw(90)
'Insufficient funds'
>>> tom_account.holder
'Tom'
```

A dual role of the object itself

1. Determines what the method's name means;
   – F.e. *withdraw* is not a name in the environment, but instead a name that is <u>local</u> to the Account class

2. Is bound to the first parameter self when the method is invoked.

Message Passing and Dot Expressions

- Methods and instance attributes are the fundamental elements of OOP.
- Behave like a ***dispatch dictionary*** with a *message passing*:
  - Objects take ***messages*** using dot notation (names local to a class)
  - Objects have named ***local state*** values (the instance attributes)
  - Local state can be accessed and manipulated using dot notation, (different!) *without **nonlocal** statements*

Dot notation and the message passing metaphor

- *Idea of Message Passing* – data values have behavior by responding to messages that are relevant to the abstract type they represent.

- *Dot notation* is a ***syntactic sugar*** that formalizes the message passing metaphor.

The advantage of a built-in object system

- Message passing can interact seamlessly with other language features, such as assignment statements:

  – different messages to “get” or “set” the value associated with a local attribute name are not required
    - a.balance = 100

  – the language syntax allows to use the message name directly
    - a.deposit(10)

Dot expressions. How to interpret?

- Consists of an expression, a dot, and a name:

  <expression> . <name>

  – **<expression>** can be any valid Python expression,
  – **<name>** must be a simple name (not an expression).

- Evaluates to the *value of the attribute* with the given *<name>,* for the *object* that is the *value of the <expression>*

>>> tom_account.balance => 90

getattr

- returns an attribute for an object by name
- equivalent of dot notation
- we can look up an attribute using a string, just as we did with a *dispatch dictionary*:

```
>>> getattr(tom_account, ’balance’)
10
```

hasattr

- tests whether an object has a named attribute:

```
>>> hasattr(tom_account, ’deposit’)
True
```

- The attributes of an object include:
  - all of its instance attributes,
  - all of the class attributes (including methods)
- Methods are attributes of the class that require special handling

Methods and functions

- When a method is invoked on an object, that object is implicitly passed as the method's <u>first argument</u>

||

- The *value of the <expression>* (left of the dot) is passed automatically as the <u>first argument</u> to the *method* (right of the dot)

- Result – the object *is bound* to the parameter ***self***.

Functions vs. bound methods

- To achieve automatic self binding, Python distinguishes between *functions* and *bound methods*

- **Bound method** couples together a *function* and the *object* on which it will be invoked

- A bound method value is associated with its first argument--the instance on which it is invoked (named *self)*--when the method is called

Functions vs. bound methods

- As an attribute of a class, a method is just a function, but as an attribute of an instance, it is a bound method:

>>> type(Account.deposit)

<class ’function’>

>>> type(tom_account.deposit)

<class ’method’>

>>> Account.deposit.__code__ == tom_account.deposit.__code__

True

a standard two-argument function with parameters self and amount

a one-argument method:
- the *self* is bound to the tom_account (automatically),
- the *amount* will be bound to the argument passed to the method

Both are associated with the same *deposit* function body

Call method

as a function

- must supply an argument for the self parameter explicitly:

>>>Account.deposit(tom_account, 1001)
The deposit function requires 2 arguments

1011

- <u>by getattr</u> with a class as its first argument:

>>> getattr(Account,'deposit')
(tom_account, 1001)

as a bound method

- the self parameter is bound automatically:

>>> tom_account.deposit(1000)
# The deposit method takes 1 argument

2011

- <u>by getattr</u> with an object as its first argument:

>> getattr(tom_account,'deposit')(1000)

Class Attributes

- Attribute values that are *shared* across all objects of a given class
- Are associated with the class itself, rather than any individual instance of the class

- Example: a bank pays interest on the balance of accounts at a fixed ***interest rate***, that is a *single value shared across all accounts*

Class attributes

- Are created by assignment statements in the suite of a class statement, *outside* of any method definition.
- May also be called *class variables* or *static variables*.

```
>>> class Account(object):
        interest = 0.02 # A class attribute
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        # Additional methods would be defined here
```

Class attributes

Can be accessed from any instance of the class:

```
>>> tom_account = Account('Tom')
>>> jim_account = Account('Jim')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
```

Assignment class attributes

A single assignment statement to a class attribute changes the value of the attribute for all instances of the class:

```
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04
>>> tom_account.interest = 0.08 #?????????????????
```

Dot expression: **evaluation**

<expression> . <name>

To evaluate a dot expression (for instance!):

1. Evaluate the <expression> to the left of the dot, which yields the **object**.
2. Match <name> against *the* ***instance attributes*** of that object; if an attribute with that name exists, its value is returned.
3. If <name> does not appear among instance attributes, then look up <name> in the ***class***, which yields a class attribute value; return its value unless it is a function.
4. If a returned value is a function, then return a ***bound method***.

Dot expression: **assignment**

- Assignment statements with a dot expr. on their left affect attributes for the *value of <expression>*:
  - If it is an *instance*, then assignment sets an instance attribute.
  - If it is a *class*, then assignment sets a class attribute.

- Consequence – assignment to an attribute of an instance cannot affect the attributes of its class!

Example

- create a new **instance attribute**:

>>> **jim_account**.interest = 0.08

- that attribute value will be returned from a dot expr.:

>>> **jim_account**.interest

0.08

- The **class attribute** *interest* still retains its *original* value (for all other accounts):

>>> **tom_account**.interest

0.04

Example

Changes to the class attribute interest will affect **tom_account**, but the instance attribute for **jim_account** will be unaffected:

>>> Account.interest = 0.05 # changing the **class attribute**

>>> tom_account.interest # changes instances without like-named instance attributes

0.05

>>> jim_account.interest # but the existing **instance attribute** is unaffected

0.08

Example

```
>>> Account.stam = 0 # add class attribute
>>> Account.stam
0
>>> Account.func = lambda x: 5 # ???
>>> Account.func(2)  # <- function
5
>>> a = Account('Sam')
>>> a.func() # <- bounded method
5
>>> Account.foo = lambda x:x   #???
```

Inheritance

- We often find that different abstract data types are related.

- Two classes may have similar attributes, but one represents a special case of the other.

Example

A checking account is different from a standard account:

1. Charges an extra $1 for each withdrawal and
2. has a lower interest rate

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20) # Deposits are the same
20
>>> ch.withdraw(5) # withdrawals decrease balance by an extra charge
14
```

A **CheckingAccount** is a specialization of an **Account**:

1. Account is the *base class* of CheckingAccount,
2. CheckingAccount is a *subclass* of Account.

is-a relationship

- A subclass inherits the attributes of its base class, but may override certain attributes, including methods.
- We only specify what is different between the subclass and the base class.
- Anything that is unspecified in the subclass is automatically assumed to behave just as it would for the base class.
- Represent **is-a** relationships between classes.
- A checking account **is-a** specific type of account, so CheckingAccount **inherits** from Account.

Using Inheritance

- We specify inheritance by putting the *base class* in parentheses after the class name

Account (base) class implementation

```
>>> class Account(object):
        """A bank account that has a non-negative balance."""
        interest = 0.02
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        def deposit(self, amount):
"""Increase the account balance by amount and return the new balance."""
            self.balance = self.balance + amount
            return self.balance
        def withdraw(self, amount):
"""Decrease the account balance by amount and return the new balance."""
            if amount > self.balance:
                return 'Insufficient funds'
            self.balance = self.balance - amount
            return self.balance
```

CheckingAccount (subclass) implementation

```
>>> class CheckingAccount(Account):
        """A bank account that charges for withdrawals."""
        withdraw_charge = 1
        interest = 0.01
        def withdraw(self, amount):                    ?
            return Account.withdraw(self, amount + self.withdraw_charge)
```

- A class attribute *withdraw_charge* is specific to the CheckingAccount class.
- We assign a lower value to the *interest* attribute.
- A new *withdraw* method overrides the behavior of the Account class.
- All other behavior is *inherited* from the base class Account.

Example

```
>>> checking = CheckingAccount('Sam')
>>> checking.deposit(10)
10
>>> checking.withdraw(5)
4
>>> checking.interest
0.01
```

evaluates to a bound method, which was defined in the **Account** class. HOW???

“looking up” a name in a class

- Python tries to find that name in *every base* class in the inheritance chain for the original object’s class.

- Recursive procedure:
  1. If it names an attribute in the class, return the attribute value.
  2. Otherwise, look up the name in the **base** class, if there is one.

Example

- In the case of ***deposit***, Python would have looked for the name:
  1. first on the *instance*,
  2. then in the *CheckingAccount class*,
  3. finally, it would look in the *Account class*, where deposit is defined.
- According to the evaluation rule for dot expressions, it evaluates to a *bound method value*.
- The method is invoked with the argument 10:
  – *self* bound to the *checking* object
  – *amount* bound to 10.

Calling ancestors

- Attributes that have been overridden are still accessible via class objects.
- Example: in the *withdraw* method of *CheckingAccount* we call the *withdraw* method of *Account* using the *withdraw_charge*.
  - We called **self.withdraw_charge** rather than the equivalent **CheckingAccount.withdraw_charge**.
  - Benefit: a class that inherits from CheckingAccount might override the withdrawal charge. We would like to find that new value instead of the old one.

Multiple Inheritance

- Python supports ***multiple inheritance*** – the concept of a subclass inheriting attributes from multiple base classes

Example

- **SavingsAccount** inherits from Account, but charges customers a small fee every time they make a deposit:

```
>>> class SavingsAccount(Account):
        deposit_charge = 2
        def deposit(self, amount):
                return Account.deposit(self, amount - self.deposit_charge)
```

Example

- **AsSeenOnTVAccount** account with the best features of both CheckingAccount and SavingsAccount: withdrawal fees, deposit fees, and a low interest rate.
- It's both a checking and a savings account in one!

```
>>> class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
        def __init__(self, account_holder):
            self.holder = account_holder
            self.balance = 1 # A free dollar!
```

Smart deal?

Both withdrawal and deposits will generate fees, using the function definitions in CheckingAccount and SavingsAccount respectively.

```
>>> such_a_deal = AsSeenOnTVAccount("John")
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20) # $2 fee from SavingsAccount.deposit
19
>>> such_a_deal.withdraw(5) # $1 fee from
    CheckingAccount.withdraw
13
```

Non-ambiguous references

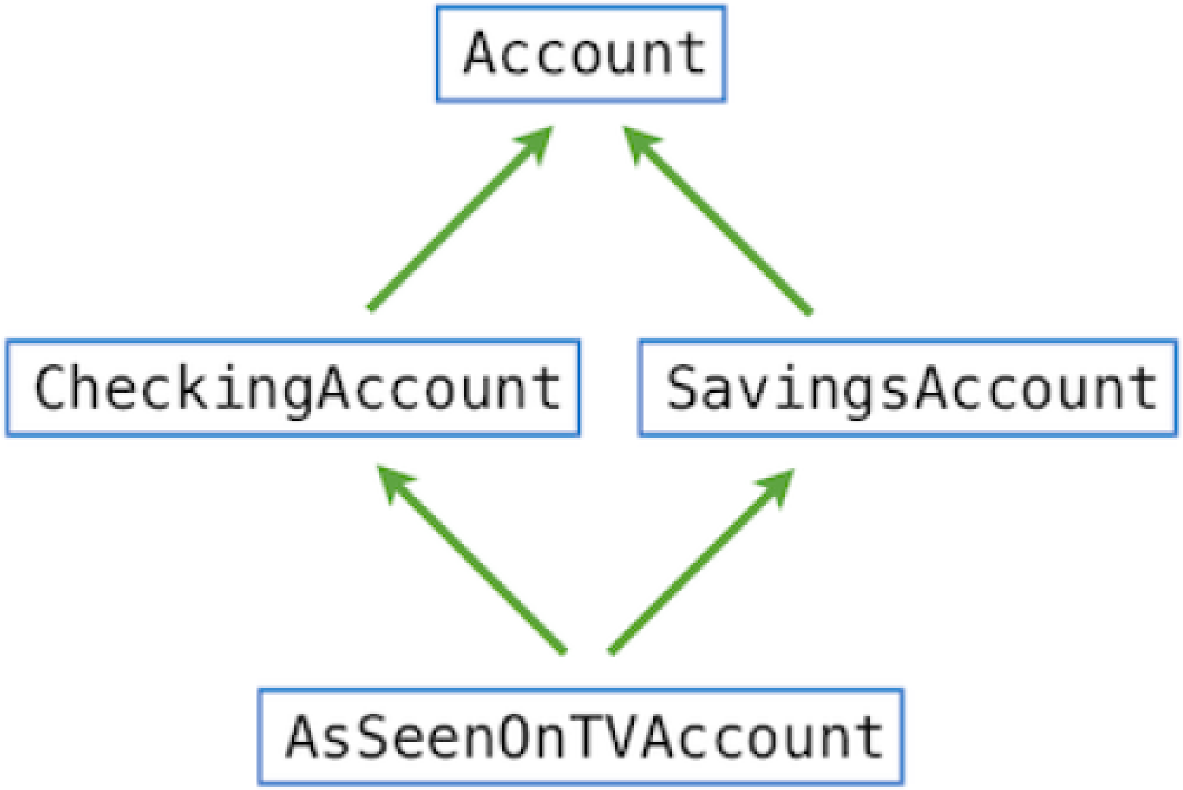
Non-ambiguous references are resolved correctly as expected:

```
>>> such_a_deal.deposit_charge
2
>>> such_a_deal.withdraw_charge
1
```

Ambiguous references

- **withdraw** method is defined in both **Account** and **CheckingAccount**
- **mortgage** method is defined in both **CheckingAccount** and **SavingAccount**
- Which code will run on such_a_deal ???
- According to the inheritance graph

Ambiguous references resolving via inheritance graph



Inheritance ordering

- For a simple “diamond” shape, Python resolves names from left to right, then upwards - BFS.
- In our example, Python checks for an attribute name in the following order:
  1. AsSeenOnTVAccount,
  2. CheckingAccount,
  3. SavingsAccount,
  4. Account,
  5. object

Inheritance ordering

- There is no correct solution to the inheritance ordering problem

- Programming language that supports multiple inheritance must select <u>some ordering (or reject)</u> in a consistent way

Even more…

- Python resolves this name using a recursive algorithm called the **C3 Method Resolution Ordering**.
- The method resolution order of any class can be queried using the **mro** method on all classes.

>>> [c.__name__ for c in AsSeenOnTVAccount.**mro()**]

['AsSeenOnTVAccount', 'CheckingAccount', 'SavingsAccount', 'Account', 'object']

The Role of Objects

- **Object system** is designed to make **data abstraction** and **message passing** both convenient and flexible.
- The *specialized syntax* of classes, methods, inheritance, and dot expressions formalizes the object metaphor, for organizing large programs.
- Each **object** in a program encapsulates and manages some part of the program's ***state***, and each **class** statement defines the functions that implement some part of the program's overall **logic**.
- **Abstraction barriers** enforce the boundaries between different aspects of a large program.

Example systems

- OOP is well-suited to programs that model systems that have separate but interacting parts.
  - Different users interact in a social network, different characters interact in a game, and different shapes interact in a physical simulation.
- When representing such systems:
  - *objects* in a program often map onto *objects* in the system,
  - *classes* represent their *types* and *relationships*