Data Structures and Algorithms

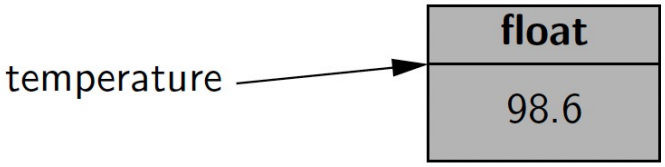
Instructor – Milan Lamichhane

Lecture 2

Objects in Python

- Python is an object-oriented language, and classes form the basis for all data types. For example, Python's built-in classes, such as the ***int*** class for integers, the ***float*** class for floating-point values, and the ***str*** class for character strings.
- Let's consider an assignment statement below

temperature = 98.6



- The identifier *temperature* references an instance of the **float** class having value 98.6.

Objects in Python

- Identifiers in Python are case-sensitive, so *temperature* and *Temperature* are distinct names.
- Identifiers can be composed of almost any combination of letters, numerals, and underscore characters.
- The primary restrictions are that an identifier cannot begin with a numeral (thus *9lives* is an illegal name).

Objects in Python

- There are 33 specially reserved words that cannot be used as identifiers.

Reserved Words								
False	as	continue	else	from	in	not	return	yield
None	assert	def	except	global	is	or	try	
True	break	del	finally	if	lambda	pass	while	
and	class	elif	for	import	nonlocal	raise	with	

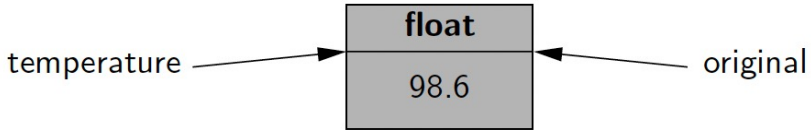
Objects in Python

- Python is a ***dynamically typed*** language, as there is no advance declaration associating an identifier with a particular data type.
- An identifier can be associated with any type of object, and it can later be reassigned to another object of the same (or different) type.
- Although an identifier has no declared type, the object to which it refers has a definite type.

Objects in Python

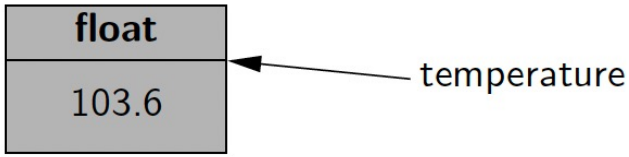
- A programmer can establish an ***alias*** by assigning a second identifier to an existing object.

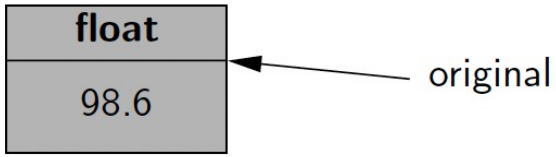
original = temperature



- if one of the names is reassigned to a new value using a subsequent assignment statement, that does not affect the aliased object, rather it breaks the alias.

temperature = temperature + 5.0





Objects in Python

- A class is ***immutable*** if each object of that class has a fixed value upon instantiation that cannot subsequently be changed.
- For example, the ***float*** class is immutable. Once an instance has been created, its value cannot be changed (although an identifier referencing that object can be reassigned to a different value).

Objects in Python

- Some of the commonly used built-in classes in Python are

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Expressions, Operators, and Precedence

Operator Precedence		
	Type	**Symbols**
1	member access	expr.member
2	function/method calls container subscripts/slices	expr(...) expr[...]
3	exponentiation	**
4	unary operators	+expr, −expr, ~expr
5	multiplication, division	*, /, //, %
6	addition, subtraction	+, −
7	bitwise shifting	<<, >>
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	|
11	comparisons containment	**is**, **is not**, ==, !=, <, <=, >, >= **in**, **not in**
12	logical-not	**not** expr
13	logical-and	**and**
14	logical-or	**or**
15	conditional	val1 **if** cond **else** val2
16	assignments	=, +=, −=, *=, etc.

Functions

- The general term ***function*** is described as a traditional, stateless function that is invoked without the context of a particular class or an instance of that class, such as ***sorted(data)***.
- Alternatively, more specific term, ***method***, is described as a member function that is invoked upon a specific object using an object-oriented message passing syntax, such as ***data.sort( )***.

Functions

- the keyword ***def***, serves as the function's signature. This establishes a new identifier as the name of the function (***count***, in this example), and it establishes the number of parameters that it expects, as well as names identifying those parameters (***data*** and ***target***, in this example).
- The remainder of the function definition is known as the ***body*** of the function.

```
def count(data, target):
  n = 0
  for item in data:
    if item == target:
      n += 1
  return n
```

Functions

- Each time a function is called, Python creates a dedicated activation record that stores information relevant to the current call. This activation record includes what is known as a ***namespace*** to manage all identifiers that have local scope within the current call.
- In the context of a function signature, the identifiers used to describe the expected parameters are known as ***formal parameters***, and the objects sent by the caller when invoking the function are the ***actual parameters***.

Functions

Consider the following call to our count function

prizes = count(grades, 'A')

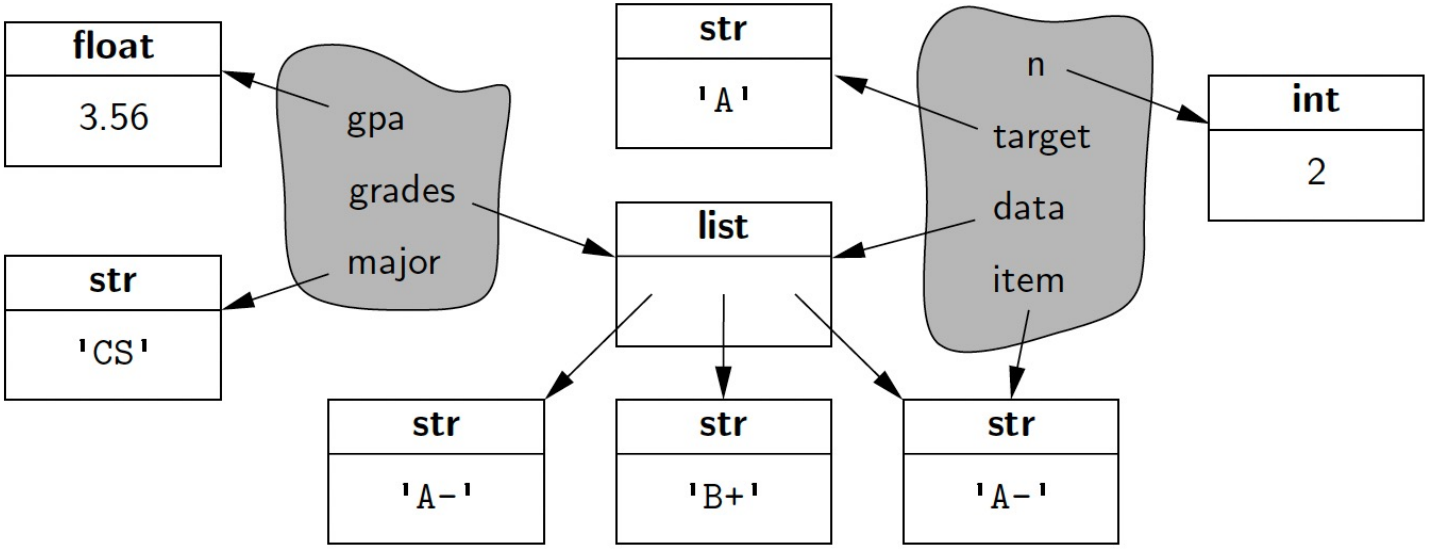
- Just before the function body is executed, the actual parameters, *grades* and *'A'*, are implicitly assigned to the formal parameters, *data* and *target*, as follows:

  data = grades

  target = 'A'

- These assignment statements establish identifier ***data*** as an alias for ***grades*** and ***target*** as a name for the string literal "***A***" .

Functions



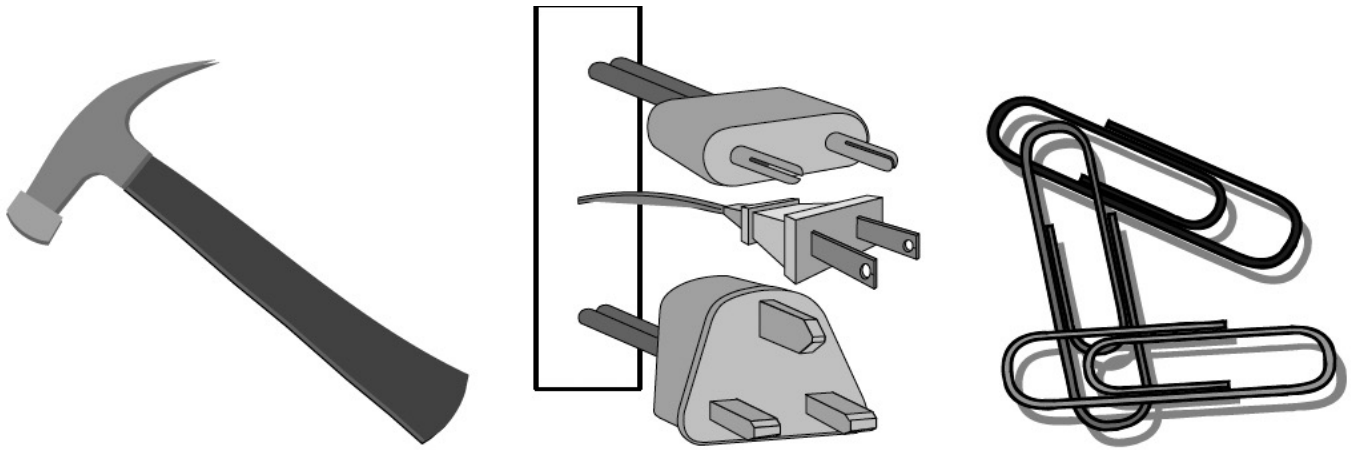
The two ***namespaces*** associated with a user's call *count(grades, 'A')*. The left namespace is the callers, and the right namespace represents the local scope of the function.

Object-Oriented Programming

- The main “actors” in the ***object-oriented*** paradigm are called objects.
- Each object is an instance of a ***class***.
- Each ***class*** presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects.
- The ***class*** definition typically specifies instance variables, also known as ***data members***, that the object contains, as well as the ***methods***, also known as ***member functions***, that the object can execute.
- Object-oriented approach of computing is intended to fulfill several goals and incorporate several design principles.

Object-Oriented Design Goals

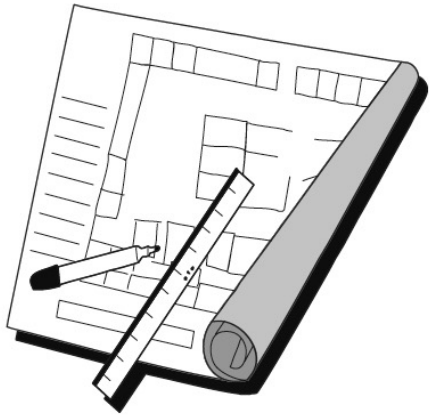
- Software implementations should achieve ***robustness***, ***adaptability***, and ***reusability***.



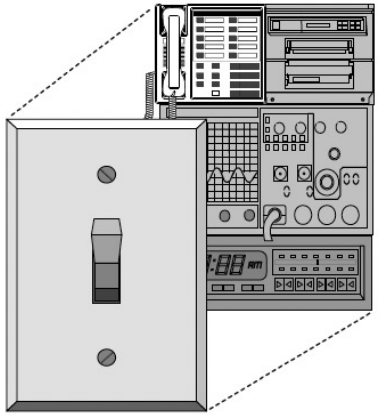
Robustness Adaptability Reusability

Object-Oriented Design Principles

Modularity



Abstraction



Encapsulation

Class Definitions

- A ***class*** serves as the primary means for abstraction in ***object-oriented programming***.
- A class provides a set of behaviors in the form of ***member functions*** (also known as ***methods***), with implementations that are common to all instances of that class.
- A ***class*** also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of ***attributes*** (also known as ***fields***, ***instance variables***, or ***data members***).

Class Definitions

- The ***self*** Identifier
  - In Python, the self identifier plays a key role.
  - Syntactically, ***self*** identifies the instance upon which a method is invoked.
  - In the context of the ***CreditCard*** class, there can presumably be many different ***CreditCard*** instances, and each must maintain its own balance, its own credit limit, and so on. Therefore, each instance stores its own instance variables to reflect its current state.

```
class CreditCard:
    """A consumer credit card."""

    def __init__(self, customer, bank, acnt, limit):
        """
        Create a new credit card instance.

        customer    the name of the customer (e.g., John Bowman )
        bank        the name of the bank (e.g., California Savings )
        acnt        the acount identifier (e.g., 5391 0375 9387 5309 )
        limit       credit limit (measured in dollars)

        The initial balance is zero.
        """
        self._customer = customer
        self._bank = bank
        self._account = acnt
        self._limit = limit
        self._balance = 0

    def get_customer(self):
        """Return name of the customer."""
        return self._customer

    def get_bank(self):
        """Return the bank s name."""
        return self._bank

    def get_account(self):...

    def get_limit(self):...

    def get_balance(self):...

    def charge(self, price):
        """Charge given price to the card, assuming sufficient credit limit.

        Return True if charge was processed; False if charge was denied.
        """
        if price + self._balance > self._limit:     # if charge would exceed limit,
            return False                            # cannot accept charge
        else:
            self._balance += price
            return True

    def make_payment(self, amount):
        """Process customer payment that reduces balance."""
        self._balance -= amount
```

Class Definitions

- A user can create an instance of the CreditCard class using a syntax as:

cc = CreditCard(John Doe, 1st Bank , 5391 0375 9387 5309 , 1000)

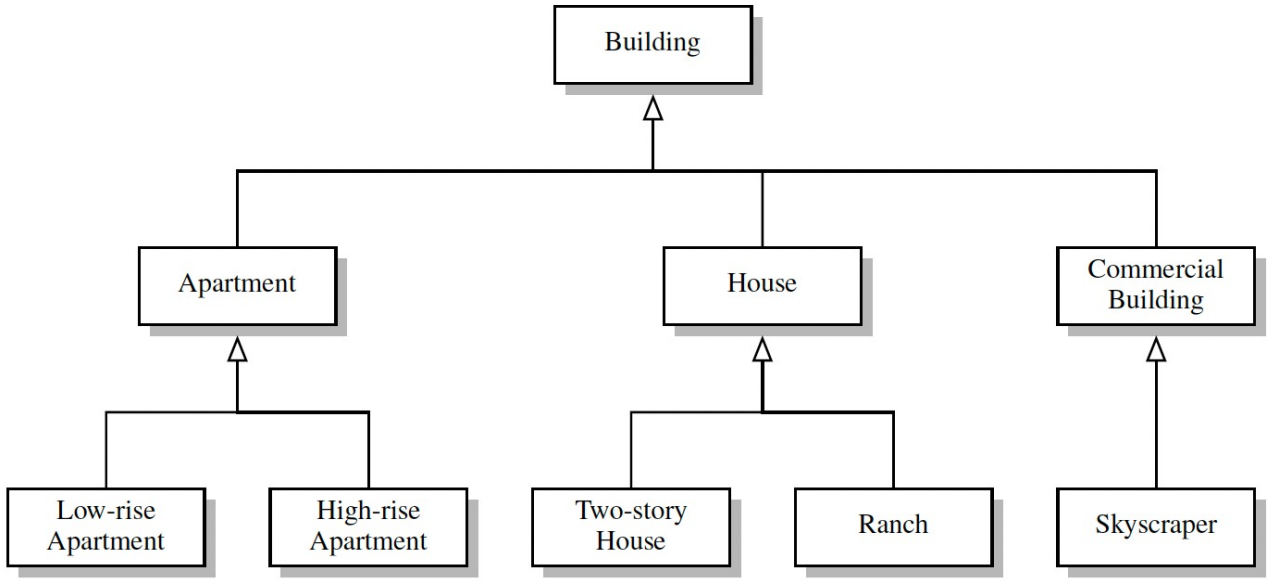
- Internally, this results in a call to the specially named ***__init__*** method that serves as the ***constructor*** of the class. Its primary responsibility is to establish the state of a newly created credit card object with appropriate instance variables.

Inheritance

- A natural way to organize various structural components of a software package is in a hierarchical fashion, with similar abstract definitions grouped together in a level-by-level manner that goes from specific to more general as one traverses up the hierarchy.
- The correspondence between levels is often referred to as an "***is a***" relationship, as a house *is a* building, and a ranch *is a* house.

Inheritance

An example of such a hierarchy is shown as below. Using mathematical notations, the set of houses is a subset of the set of buildings, but a superset of the set of ranches.



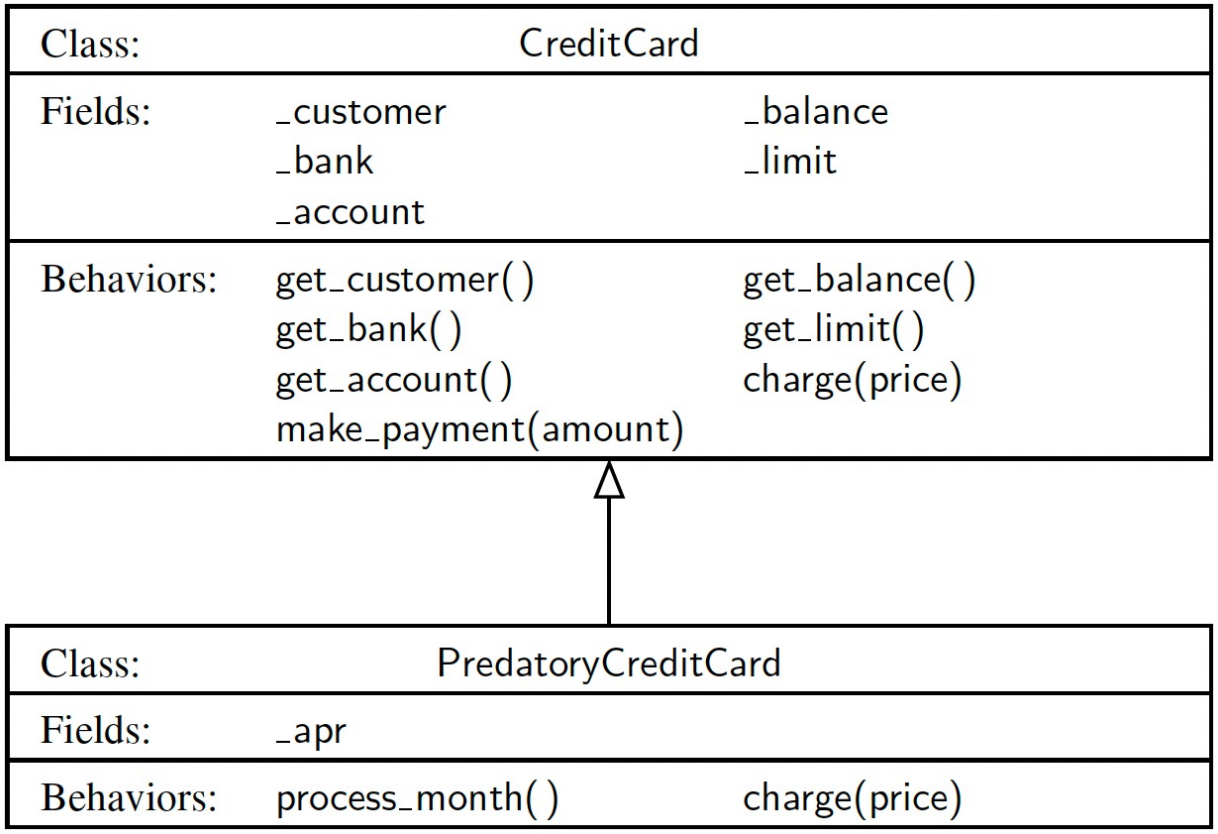
Inheritance

- In object-oriented programming, the mechanism for a modular and hierarchical organization is a technique known as inheritance.
- This allows a new class to be defined based upon an existing class as the starting point.
- In object-oriented terminology, the existing class is typically described as the ***base class***, ***parent class***, or ***superclass***, while the newly defined class is known as the ***subclass*** or ***child class***.

Inheritance

- There are two ways in which a subclass can differentiate itself from its superclass.
  - A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method.
  - A subclass may also extend its superclass by providing brand new methods.

Inheritance



Inheritance

- The new class will differ from the original in two ways:
  - If an attempted charge is rejected because it would have exceeded the credit limit, a $5 fee will be charged
  - There will be a mechanism for assessing a monthly interest charge on the outstanding balance, based upon an Annual Percentage Rate (APR) specified as a constructor parameter.
- To charge a fee for an invalid charge attempt, we ***override*** the existing charge method, thereby specializing it to provide the new functionality.

```
import CreditCard

class PredatoryCreditCard(CreditCard):
    """An extension to CreditCard that compounds interest and fees."""

    def __init__(self, customer, bank, acnt, limit, apr):
        """Create a new predatory credit card instance.

        The initial balance is zero.

        customer    the name of the customer (e.g., John Bowman )
        bank        the name of the bank (e.g., California Savings )
        acnt        the acount identifier (e.g., 5391 0375 9387 5309 )
        limit       credit limit (measured in dollars)
        apr         annual percentage rate (e.g., 0.0825 for 8.25% APR)
        """
        super().__init__(customer, bank, acnt, limit) # call super constructor
        self._apr = apr

    def charge(self, price):
        """Charge given price to the card, assuming sufficient credit limit.

        Return True if charge was processed.
        Return False and assess 5 fee if charge is denied.
        """
        success = super().charge(price)     # call inherited method
        if not success:
            self._balance += 5              # assess penalty
        return success                      # caller expects return value

    def process_month(self):
        """Assess monthly interest on outstanding balance."""
        if self._balance > 0:
        # if positive balance, convert APR to monthly multiplicative factor
            monthly_factor = pow(1 + self._apr, 1/12)
            self._balance = monthly_factor
```

Operator Overloading

- Python's built-in classes provide natural semantics for many operators.
- For example, the syntax a + b invokes addition for numeric types, yet concatenation for sequence types.
- When defining a new class, we must consider whether a syntax like ***a + b*** should be defined when a or b is an instance of that class.
- By default, the + operator is undefined for a new class. However, the author of a class may provide a definition using a technique known as operator overloading.
- This is done by implementing a specially ***named method***.

Operator Overloading

- In particular, the + operator is overloaded by implementing a method named __***add***__, which takes the right-hand operand as a parameter, and which returns the result of the expression.
- That is, the syntax, ***a + b***, is converted to a method call on object a of the form, ***a.__add__(b)***.
- Similar specially ***named methods*** exist for other operators.

Operator Overloading

Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a * b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ** b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)

Assignment

- Write a class called ***Rectangle*** which contains the method to return the area of a rectangle object. Also, this class should overload an operator to compare the rectangles based on their area.