Object-Oriented Programming

Terminology

- Each **object** created in a program 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.

Goals

- Robustness
  - We want software to be capable of handling unexpected inputs that are not explicitly defined for its application.
- Adaptability
  - Software needs to be able to evolve over time in response to changing conditions in its environment.
- Reusability
  - The same code should be usable as a component of different systems in various applications.

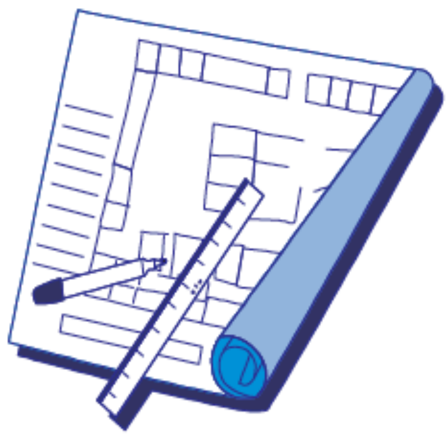
Abstract Data Types

- **Abstraction** is to distill a system to its most fundamental parts.
- Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs).
- An ADT is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations.
- An ADT specifies what each operation does, but not how it does it.
- The collective set of behaviors supported by an ADT is its **public interface**.

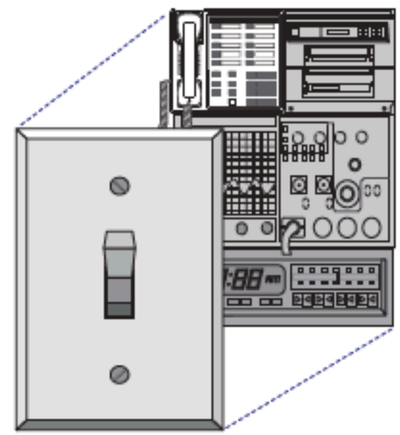
Object-Oriented Design Principles

- Modularity
- Abstraction
- Encapsulation

Modularity

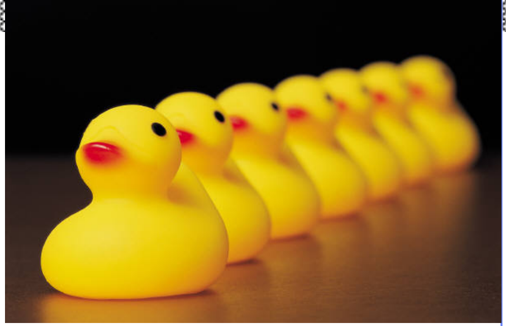


Abstraction



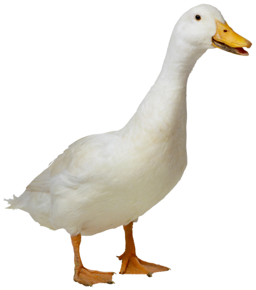
Encapsulation

Duck Typing



- Python treats abstractions implicitly using a mechanism known as **duck typing**.
  - A program can treat objects as having certain functionality and they will behave correctly provided those objects provide this expected functionality.
- As an interpreted and dynamically typed language, there is no "compile time" checking of data types in Python, and no formal requirement for declarations of abstract base classes.
- The term "duck typing" comes from an adage attributed to poet James Whitcomb Riley, stating that "when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

Abstract Base Classes



- Python supports abstract data types using a mechanism known as an **abstract base class (ABC)**.
- An abstract base class cannot be instantiated, but it defines one or more common methods that all implementations of the abstraction must have.
- An ABC is realized by one or more concrete classes that inherit from the abstract base class while providing implementations for those method declared by the ABC.
- We can make use of several existing abstract base classes coming from Python's collections module, which includes definitions for several common data structure ADTs, and concrete implementations of some of these.

Encapsulation

- Another important principle of object-oriented design is **encapsulation**.
  - Different components of a software system should not reveal the internal details of their respective implementations.
- Some aspects of a data structure are assumed to be public and some others are intended to be internal details.
- Python provides only loose support for encapsulation.
  - By convention, names of members of a class (both data members and member functions) that start with a single underscore character (e.g., _secret) are assumed to be nonpublic and should not be relied upon.

Design Patterns

- **Algorithmic patterns:**
- Recursion
- Amortization
- Divide-and-conquer
- Prune-and-search
- Brute force
- Dynamic programming
- The greedy method

- **Software design patterns:**
- Iterator
- Adapter
- Position
- Composition
- Template method
- Locator
- Factory method

Object-Oriented Software Design

- **Responsibilities**: Divide the work into different actors, each with a different responsibility.
- **Independence**: Define the work for each class to be as independent from other classes as possible.
- **Behaviors**: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.

Unified Modeling Language (UML)

A **class diagram** has three portions.

1. The name of the class
2. The recommended instance variables
3. The recommended methods of the class.

Class:	CreditCard	
Fields:	_customer _bank _account	_balance _limit
Behaviors:	get_customer( ) get_bank( ) get_account( ) make_payment(amount)	get_balance( ) get_limit( ) charge(price)

Class Definitions

- A class serves as the primary means for abstraction in object-oriented programming.
- In Python, every piece of data is represented as an instance of some class.
- A class provides a set of behaviors in the form of member functions (also known as **methods**), with implementations that belong to all its instances.
- 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**).

The **self** Identifier

- In Python, the **self** identifier plays a key role.
- In any class, there can possibly be many different instances, and each must maintain its own instance variables.
- Therefore, each instance stores its own instance variables to reflect its current state. Syntactically, **self** identifies the instance upon which a method is invoked.

Example

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

  def __init__(self, customer, bank, acnt, limit):
    """Create a new 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)
    """
    self._customer = customer
    self._bank = bank
    self._account = acnt
    self._limit = limit
    self._balance = 0

```

Example, Part 2

```
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):
    """Return the card identifying number (typically stored as a string)."""
    return self._account

def get_limit(self):
    """Return current credit limit."""
    return self._limit

def get_balance(self):
    """Return current balance."""
    return self._balance
```

Example, Part 3

```
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
```

Constructors

- 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.

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.

Iterators

- Iteration is an important concept in the design of data structures.
- An **iterator** for a collection provides one key behavior:
  - It supports a special method named `__next__` that returns the next element of the collection, if any, or raises a StopIteration exception to indicate that there are no further elements.

Automatic Iterators

- Python also helps by providing an automatic iterator implementation for any class that defines both `__len__` and `__getitem__`.

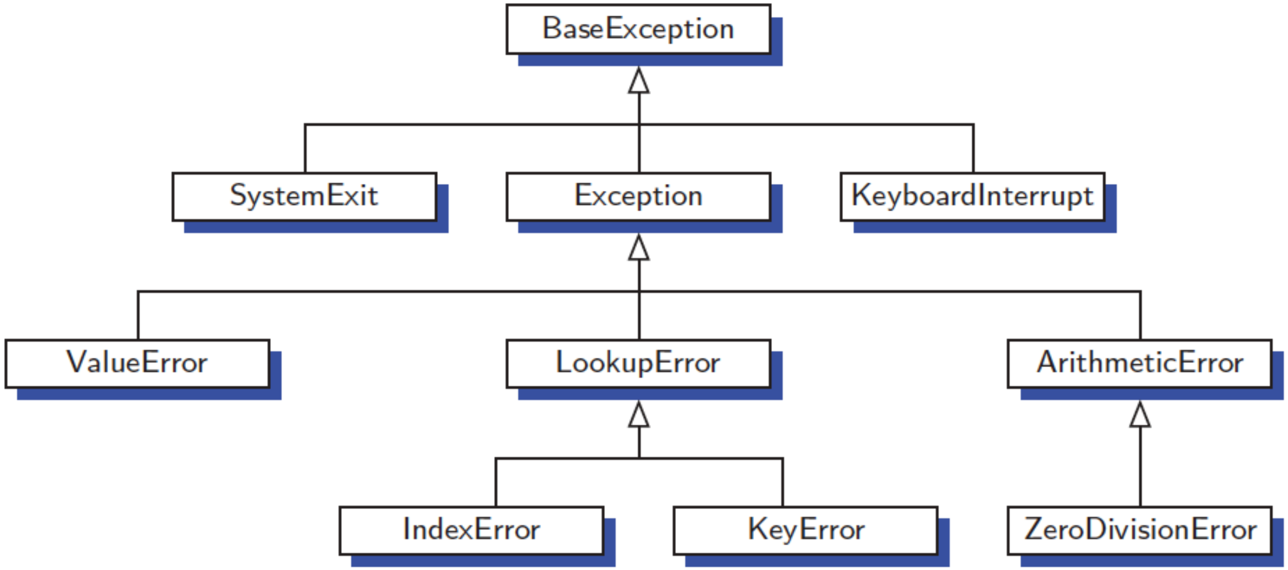
```
class Range:
  """A class that mimic's the built-in range class."""

  def __init__(self, start, stop=None, step=1):
    """Initialize a Range instance.

    Semantics is similar to built-in range class.
    """
    if step == 0:
      raise ValueError('step cannot be 0')

    if stop is None:                  # special case of range(n)
      start, stop = 0, start          # should be treated as if range(0,n)

    # calculate the effective length once
    self._length = max(0, (stop − start + step − 1) // step)

    # need knowledge of start and step (but not stop) to support __getitem__
    self._start = start
    self._step = step

  def __len__(self):
    """Return number of entries in the range."""
    return self._length

  def __getitem__(self, k):
    """Return entry at index k (using standard interpretation if negative)."""
    if k < 0:
      k += len(self)                  # attempt to convert negative index

    if not 0 <= k < self._length:
      raise IndexError('index out of range')

    return self._start + k * self._step
```

Inheritance

- A mechanism for a modular and hierarchical organization is **inheritance**.
- This allows a new class to be defined based upon an existing class as the starting point.
- 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.
- 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.

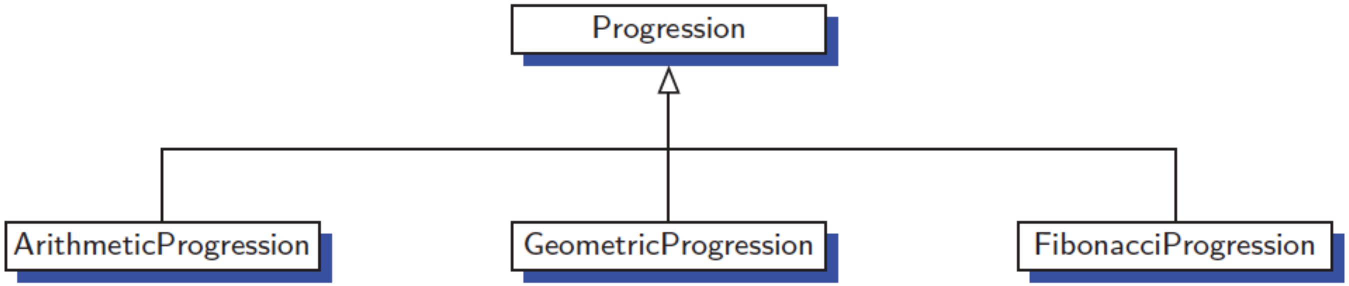
Inheritance is Built into Python

- A portion of Python's hierarchy of exception types:



An Extended Example

- A **numeric progression** is a sequence of numbers, where each number depends on one or more of the previous numbers.
  - An **arithmetic progression** determines the next number by adding a fixed constant to the previous value.
  - A **geometric progression** determines the next number by multiplying the previous value by a fixed constant.
  - A **Fibonacci progression** uses the formula $N_{i+1}=N_i+N_{i-1}$



The Progression Base Class

```
class Progression:
  """Iterator producing a generic progression.

  Default iterator produces the whole numbers 0, 1, 2, ...
  """

  def __init__(self, start=0):
    """Initialize current to the first value of the progression."""
    self._current = start

  def _advance(self):
    """Update self._current to a new value.

    This should be overridden by a subclass to customize progression.

    By convention, if current is set to None, this designates the
    end of a finite progression.
    """
    self._current += 1

  def __next__(self):
    """Return the next element, or else raise StopIteration error."""
    if self._current is None:       # our convention to end a progression
      raise StopIteration()
    else:
      answer = self._current        # record current value to return
      self._advance()               # advance to prepare for next time
      return answer                 # return the answer

  def __iter__(self):
    """By convention, an iterator must return itself as an iterator."""
    return self

  def print_progression(self, n):
    """Print next n values of the progression."""
    print(' '.join(str(next(self)) for j in range(n)))
```

ArithmeticProgression Subclass

```
class ArithmeticProgression(Progression):          # inherit from Progression
  """Iterator producing an arithmetic progression."""

  def __init__(self, increment=1, start=0):
    """Create a new arithmetic progression.

    increment   the fixed constant to add to each term (default 1)
    start       the first term of the progression (default 0)
    """
    super().__init__(start)                         # initialize base class
    self._increment = increment

  def _advance(self):                               # override inherited version
    """Update current value by adding the fixed increment."""
    self._current += self._increment
```

GeometricProgression Subclass

```
class GeometricProgression(Progression):           # inherit from Progression
  """Iterator producing a geometric progression."""

  def __init__(self, base=2, start=1):
    """Create a new geometric progression.

    base       the fixed constant multiplied to each term (default 2)
    start      the first term of the progression (default 1)
    """
    super().__init__(start)
    self._base = base

  def _advance(self):                              # override inherited version
    """Update current value by multiplying it by the base value."""
    self._current *= self._base
```

FibonacciProgression Subclass

```
class FibonacciProgression(Progression):
  """Iterator producing a generalized Fibonacci progression."""

  def __init__(self, first=0, second=1):
    """Create a new fibonacci progression.

    first       the first term of the progression (default 0)
    second      the second term of the progression (default 1)
    """
    super().__init__(first)                 # start progression at first
    self._prev = second − first             # fictitious value preceding the first

  def _advance(self):
    """Update current value by taking sum of previous two."""
    self._prev, self._current = self._current, self._prev + self._current
```