Programming for Business Computing

Classes: A Motivating Example

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Motivating example: dates

- In many applications we deal with dates.
  - Suppose that we do not know the **`datetime`** library.
- A date is consist of three attributes.
  - Year: an integer from 1 to 3000.
  - Month: an integer from 1 to 12.
  - Day: an integer from 1 to 31 (depending on month).
- If we want to store the birthdays of a group of students, what should we do?

Birthday dictionary

```
bdDict = dict()
while True:
  name = input("name: ")
  if name == "":
    break

  birthday = input("birthday (yyyy/mm/dd): ")
  if birthday == "":
    break

  bdDict[name] = birthday

print(bdDict)
```

- How to prevent a date like 2016/14/20 or 2015/09/31?
  - Be aware of leap years!

Is it a leap year?

- A year is a leap year if:
  - It is a multiple of 4 and not a multiple of 100.
  - It is a multiple of 400.

```
def isLeap(year):
  if year % 400 == 0:
    return True
  elif (year % 4 == 0) and (year % 100 != 0):
    return True
  else:
    return False
```

Is it a valid date?

```python
def isValidDate(birthday): # birthday is a yyyy/mm/dd string
  year, month, day = birthday.split("/")
  year = int(year)
  month = int(month)
  day = int(day)

  if (1 <= year <= 3000) and (1 <= month <= 12):
    daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    if isLeap(year) == True:
      daysInMonth[1] = 29
    if 1 <= day <= daysInMonth[month - 1]:
      return True
  return False
```

Fail-safe birthday dictionary

```
bdDict = dict()
while True:
  name = input("name: ")
  if name == "":
    break

  birthday = input("birthday (yyyy/mm/dd): ")

  if isValidDate(birthday) == True:
    bdDict[name] = birthday
  else:
    print("bad date!")

  if birthday == "":
    break

print(bdDict)
```

What if…

- This is good, but what if we want to know:
  - The number of people born in a given year?
  - The names of people born in a given month?
- It would be better (in many cases) if we store **three integers** instead of a string.
  - Especially when the above operations must be done frequently.
- Option 1: Three dictionaries whose keys are names and values are years, months, and days, respectively.
  - It is hard to use and easy to be **inconsistent**.
- Option 2: One dictionary whose key is name and value is a three-dimensional list (or tuple or dictionary).
  - It is non-intuitive and/or **inefficient**.
- Is there a more intuitive way?

Programming for Business Computing

Classes: Basics

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Self-defined data types: class

- It is all about data types.
  - We have basic data types like character, integer, float, Boolean, etc.
  - We have composite date types like string, list, tuple, dictionary, etc.
  - May we define a new data type to store dates?
- In Python, we define our own data type by defining a **class**.
  - A class can be used to declare **objects** (variables whose type is a class).
  - For each object, we define attributes called **instance variables**.
  - For each class, we define attributes called **static variables**.
  - An attribute is also called a **member** of an object or a class.

Defining a class and declaring an object

- To define a class, we use the keyword **`class`**:

```
class Date:
  pass
```

  - We do not have anything to be declared in the class now.
  - **`pass`** means do nothing.
- Then we may use the class to declare objects:
  - We use the **dot operator** to access a member:

```
d = Date()
print(d.month)
print(d.day)
print(d) # what is this?
```

Adding attributes to an object

- We may add new attributes to an object by **declaring an instance variable**.

```
d = Date()
d.month = 12
d.day = 31
print(d.month, d.day)
```

- Objects of the same class may have different members:

```
d2 = Date()
d2.day = 31
d2.weekday = "Mon"
print(d2.day, d2.weekday)
```

- Do not do this!
  - Unless you really know what you are doing.

Why classes and objects?

- The most obvious reason of using classes and objects is to **group multiple variables into one variable**.
  - Each variable has its **variable name**.
- Recall our birthday dictionary example and our hope to store three integers.
  - Option 1: Three dictionaries whose keys are names and values are years, months, and days, respectively.
  - Option 2: One dictionary whose key is name and value is a three-dimensional list (or tuple or dictionary).
  - Option 3: One dictionary whose key is name (a string) and value is birthday (a **`Date`**).
- Let's revise our program with the class **`Date`**.

Revising the birthday dictionary (1/4)

```python
class Date: # the basic setting
  pass

def isLeap(year): # not changed
  if year % 400 == 0:
    return True
  elif (year % 4 == 0) and (year % 100 != 0):
    return True
  else:
    return False
```

Revising the birthday dictionary (2/4)

```
def isValidDate(bDay): # bDay is a Date object
  if (1 <= bDay.year <= 3000) and (1 <= bDay.month <= 12):
    daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    if isLeap(bDay.year) == True:
      daysInMonth[1] = 29
    if 1 <= bDay.day <= daysInMonth[bDay.month - 1]:
      return True
  return False

def strToDate(birthday): # birthday is a yyyy/mm/dd string
  d = Date()
  year, month, day = birthday.split("/")
  d.year = int(year)
  d.month = int(month)
  d.day = int(day)
  return d
```

Revising the birthday dictionary (3/4)

```
bdDict = dict()
while True:
  name = input("name: ")
  if(name == ""):
    break

  birthday = input("birthday (yyyy/mm/dd): ") # birthday is a string
  birthday = strToDate(birthday) # now birthday is a Date

  if isValidDate(birthday) == True:
    bdDict[name] = birthday # now the value of a dictionary entry is a Date
  else:
    print("bad date!")

  if birthday == "":
    break

print(bdDict) # what will be printed out?
```

Revising the birthday dictionary (4/4)

```
def toString(bDay): # bDay is a Date object
  return str(bDay.year) + "/" + str(bDay.month) + "/" + str(bDay.day)

def printBdayDict(bdDict):
  for p in bdDict.keys():
    b = bdDict[p] # b is a Date object
    print(p + " was born at " + toString(b))

bdDict = dict()
while True:
  // omitted; see previous page

printBdayDict(bdDict)
```

Summary

- In short:
  - An object is a collection of variables (sometimes objects).
  - Moreover, these variables have names.
  - More benefits are to be introduced.

Programming for Business Computing

Classes: Instance functions

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Instance functions of `Date`

- Recall our program with five global functions:

```
def isLeap(year):
def isValidDate(bDay):
def strToDate(birthday):
def toString(bDay):
def printBdayDict(bdDict):
```

- In the current design, **`isValidDate()`** and **`toString()`** are like two **machines**. They do something on a **`Date`** object and return something.
  - In an alternative design, we say **`isValidDate()`** and **`toString()`** are two **operations** attached on **`Date`** objects. Each **`Date`** object has these two operations that they may do, just like having those attributes.
  - This is an important programming design philosophy: **object-oriented programming** (**OOP**).

Object-oriented programming

- Classes can be used to **modularize** programs.
- In our first attempt, we group attributes into objects/classes.
- We may also group **operations** into objects/classes:
  - We will implement **instance functions** for operations.
  - There are also **static functions**.
  - Instance functions and static functions are both called **member functions**.

Invoking instance functions

- To invoke an instance function, we also use the **dot operator**.
- When we invoke an object's instance function, we call the object the **invoking object** and the function the **invoked instance function**.

```
bdDict = dict()

while True:
  name = input("name: ")
  if(name == ""):
    break

  birthday = input("birthday (yyyy/mm/dd): ")
  birthday = strToDate(birthday)

  if(birthday.isValidDate() == True):
    bdDict[name] = birthday
  else:
    print("bad date!")

  if(birthday == ""):
    break

printBdayDict(bdDict)
```

Instance function: `isValidDate()`

- Let’s start by defining **`isValidDate()`**.

```
class Date:
  def isValidDate(self): # the invoker is a Date object
    if((1 <= self.year <= 3000) and (1 <= self.month <= 12)):
      daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    if(isLeap(self.year) == True):
      daysInMonth[1] = 29
    if(1 <= self.day <= daysInMonth[self.month - 1]):
      return True
    return False
```

- We define a function **`isValidDate()`** inside the class **`Date`**.
- An instance function’s **first parameter** is always the **invoking object** itself.
  - Through this parameter, we access the invoking object’s instance variables and instance functions.
- It does not need to be named as “self.”

Instance function: `toString()`

- Let’s convert the global function **`toString()`** into an instance one.
- Originally:

```
def toString(bDay): # bDay is a Date object
  return str(bDay.year) + "/" + str(bDay.month) + "/" + str(bDay.day)

def printBdayDict(bdDict):
  for p in bdDict.keys():
    b = bdDict[p]
    print(p + " was born at " + toString(b))
```

Instance function: `toString()`

- Now:

```
class Date:
  # others omitted
  def toString(self):
    return str(self.year) + "/" + str(self.month) + "/" + str(self.day)

def printBdayDict(bdDict):
  for p in bdDict.keys():
    b = bdDict[p]
    print(p + " was born at " + b.toString())
```

- Should we also convert **`printBdayDict()`** into an instance function?
  - No, because it should not be an operation of a **`Date`** object.

Instance function: **isLeap()**

- How about **isLeap()**? Should it be an instance function?

```
def isLeap(year):
  if(year % 400 == 0):
    return True
  elif((year % 4 == 0) and (year % 100 != 0)):
    return True
  else:
    return False
```

- It is also not an operation of a **Date** object.
- However, it is natural for a **Date** object to check **whether it is in a leap year**:
  - Just input its year attribute to **isLeap()**!

Instance function: `isLeap()`

- Let’s do it:

```
class Date:
  # others omitted

  def isValidDate(self): # the invoker is a Date object
    if((1 <= self.year <= 3000) and (1 <= self.month <= 12)):
      daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    if(self.isLeap() == True):
      daysInMonth[1] = 29
    if(1 <= self.day <= daysInMonth[self.month - 1]):
      return True
    return False

  def isLeap(self):
    return isLeap(self.year)
```

Invoking an instance function

- Pay attention to the way of invoking a instance function!

```
birthday.isValidDate() # good
isValidDate(birthday) # syntax error
birthday.isValidDate(birthday) # syntax error
```

- If we execute **`isValidDate(birthday)`**:
  - "NameError: name 'isValidDate' is not defined"
  - The function is an **instance function** (which belongs to a class), not a **global function** (which belongs to everyone).
- If we execute **`birthday.isValidDate(birthday)`**:
  - "TypeError: isValidDate() takes exactly 1 argument (2 given)"
  - The invoking object is considered as the first argument (even if there is nothing inside the pair of parentheses).

Instance functions with more parameters

- We may also have instance functions with more parameters.
- For example, how to compare whether one date is later than the other date?
- We may implement an instance function **`isLaterThan()`** for **`Date`** and input another **`Date`** object as an argument.

```
class Date:
  def isLaterThan(self, aDate):
    if self.year > aDate.year:
      return True
    elif self.year == aDate.year:
      if self.month > aDate.month:
        return True
      elif self.month == aDate.month:
        if self.day > aDate.day:
          return True
    return False

  def toString(self):
    return str(self.year) + "/" + str(self.month) + "/" + str(self.day)
```

Instance functions with more parameters

- Let’s try it:

```
def strToDate(birthday): # birthday is a yyyy/mm/dd string
  d = Date()
  year, month, day = birthday.split("/")
  d.year = int(year)
  d.month = int(month)
  d.day = int(day)
  return d

day1 = input("yyyy/mm/dd: ")
day1 = strToDate(day1)

day2 = input("yyyy/mm/dd: ")
day2 = strToDate(day2)

if day1.isLaterThan(day2):
  print(day1.toString() + " is later than " + day2.toString())
else:
  print(day1.toString() + " is note later than " + day2.toString())
```

Instance functions vs. global functions

- Note that we may also implement a **global function** with two **`Date`** objects as its two parameters.
- The previous program will then becomes something like

```
if isLaterThan(day1, day2):
  print(day1.toString() + " is later than " + day2.toString())
else:
  print(day1.toString() + " is note later than " + day2.toString())
```

- Which one do you prefer?

Programming for Business Computing

Classes: Advance topics

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Instance function: `init()`

- We want to make sure that all **`Date`** objects have the required attributes.

```
d = Date()
d.year = 2018
d.month = 4
d.isValidDate() # run-time error!
```

- Let’s write a member function that initializes member variables based on input arguments.

Member function: `init()`

- Implementation:

```
class Date:
  # others omitted

  def init(self, year, month, day):
    self.year = year
    self.month = month
    self.day = day

d = Date()
d.init(2018, 4, 5)
d.isValidDate()    # fine!
d2 = Date()
d2.isValidDate()   # run-time error!
```

  - Note how to distinguish an instance variable and a parameter.
  - Note that we **cannot** force one to invoke **`init()`**.

Constructor: `__init__()`

- Even though we have defined **`init()`**, we cannot force one (ourselves) to invoke it.
  - We may play with **uninitialized objects**.
- To resolve this issue, Python (and many other languages) allows us to define a **constructor** for a class.
  - In Python, it is a member function called **`__init__()`**.
  - It is automatically invoked when an object is created.
  - The correct number of arguments must be prepared when creating an object.

Constructor: __init__()

- Implementation:

```
class Date:
  # others omitted

  def __init__(self, year, month, day):
    self.year = year
    self.month = month
    self.day = day

d = Date(2018, 4, 5)
d.isValidDate()      # fine!
d2 = Date()          # run-time error!
```

Printing out an object directly

- Recall the function **toString()**, which returns a string of the current status.
  - We may print out the returned string.
  - However, we “cannot” **print out the object** directly.

```
class Date:
  # others omitted

 def toString(self):
    return str(self.year) + "/" + str(self.month) + "/" + str(self.day)

d = Date(2018, 4, 5)
print(d.toString())  # 2018/4/5
print(d)             # What is this?
```

A special function: `__str__()`

- We may define a new function `__str__()`:

```
class Date:
  # others omitted

 def toString(self):
    return str(self.year) + "/" + str(self.month) + "/" + str(self.day)

 def __str__(self):
    return self.toString()

d = Date(2018, 4, 5)
print(d.toString())  # 2018/4/5
print(d)             # 2018/4/5
```

- When we print out an object:
  - If a function named **`__str__()`** is defined, its returned value is printed out.
  - Otherwise, the memory information is printed out.

Two ways of printing out a date

- Suppose that in our program there are two ways of printing out April 5, 2018:
  - Single-digit if possible: 2018/4/5.
  - Always double-digit: 2018/04/05.
- We want to have the flexibility:
  - Maybe we may add a Boolean variable **`doubleDigit`** into a **`Date`** object.
  - We may then implement **`toString()`** with **`doubleDigit`**.

Two ways of printing out a date

- Let’s try it:

```python
class Date:
  # others omitted

  def __init__(self, year, month, day, doubleDigit):
    self.year = year
    self.month = month
    self.day = day
    self.doubleDigit = doubleDigit

  def toString(self):
    if self.doubleDigit == False:
      return str(self.year) + "/" + str(self.month) + "/" + str(self.day)
    else:
      dateStr = str(self.year) + "/"
      dateStr += "0" + str(self.month) if self.month < 10 else str(self.month)
      dateStr += "/"
      dateStr += "0" + str(self.day) if self.day < 10 else str(self.day)
      return dateStr
```

Static variables

- In some cases, an attribute (or property) should belong to a class and is shared by all objects of that class.
  - `doubleDigit` is such an attribute.
- We declare **static variables** for **class-specific** attributes.
  - Just like declaring **instance variables** for **object-specific** attributes.
- Both static variables and instance variables are called **member variables**.

Static variables

- To declare a static variable, do it at the place of declaring instance functions:

```
class Date:
  doubleDigit = False

  def toString(self):
    if Date.doubleDigit == False:
      return str(self.year) + "/" + str(self.month) + "/" + str(self.day)
    else:
      dateStr = str(self.year) + "/"
      dateStr += "0" + str(self.month) if self.month < 10 else str(self.month)
      dateStr += "/"
      dateStr += "0" + str(self.day) if self.day < 10 else str(self.day)
      return dateStr
```

Static variables

- To declare a static variable, do it at the place of declaring instance functions:

```
d = Date(2018, 4, 5)
print(d.toString())     # 2018/4/5
Date.doubleDigit = True
d2 = Date(2018, 4, 5)
print(d2.toString())    # 2018/04/05
```

  – Once **`Date.doubleDigit`** is set to a value, all the following invocations of **`toString()`** and **`__str__()`** from all **`Date`** objects use that value.

Static variables

- How about this?

```
d = Date(2018, 4, 5)
print(d.toString())        # 2018/4/5
doubleDigit = True
d2 = Date(2018, 4, 5)
print(d2.toString())       # 2018/4/5
```

- How about this?

```
d = Date(2018, 4, 5)
print(d.toString())        # 2018/4/5
d.doubleDigit = True
d2 = Date(2018, 4, 5)
print(d2.toString())       # 2018/4/5
```

Static functions

- There are also **static functions**.
  - They belong to the class rather than an object.
  - They are defined in the class with no invoking object (as a parameter).
  - They cannot access instance variables.
  - Static functions and instance functions are both called **member functions**.
- Let’s create a static function to modify `doubleDigit`:
  - `@staticmethod` is a **decorator**.
  - `d.setDoubleDigit(True)` also works, but we **should not** do this.

```
class Date:
  doubleDigit = False

  @staticmethod
  def setDoubleDigit(dd):
    Date.doubleDigit = dd

d = Date(2018, 4, 5)
print(d.toString())
Date.setDoubleDigit(True)
d2 = Date(2018, 4, 5)
print(d2.toString())
```