JTSK-350112

Advanced Programming in Python

Python II

Lecture 1 & 2

Dr. Kinga Lipskoch

Spring 2018

Who am I?

- PhD in Computer Science at the Carl von Ossietzky University of Oldenburg
- University lecturer at the Computer Science Department
- Joined Jacobs University in January 2013
- Office: Research I, Room 94
- Telephone: +49 421 200-3148
- E-Mail: k.lipskoch@jacobs-university.de
- Office hours: Mondays 10:00 – 12:00

Course Goals

- Learn more aspects of procedural and object-oriented programming
- Learn more details of the Python programming language
- Write and test more advanced programs
- "Hands on": not just theory, but practice sessions to apply what you have learned in the lectures

Course Details

- 3 weeks (24 hours)
- Every week will consist of:
  - 2 lectures per week (Thu/Fri afternoon, 14:15 – 16:00)
  - 2 lab sessions per week (Thu/Fri afternoon, 16:15 – 18:30)
- During each lab session you will have to solve a programming assignment sheet with multiple problems related to the previous lecture

Course Resources

- Slides, assignments and general purpose info will be posted on page of course (via Grader https://grader.eecs.jacobs-university.de/courses/350112/2018_1gB/index.html)
- "Open door" policy: feel free to knock and ask for additional explanations when needed
- Do not hesitate, and do not wait until you are left behind

Sources for These Slides and Additional Resources

- Kenneth A. Lambert: Fundamentals of Python Data Structures, Cengage Learning PTR, 2014
- Mark Summerfield: Programming in Python : A complete introduction to the Python language, second edition, Pearson Education, 2010
- John Zelle: Python Programming: An introduction to Computer Science, second edition, Franklin, Beedle & Associates, 2009
- Igor Milovanovic: Python Data Visualization Cookbook, Packt Publishing, 2013

Grading Policy

- Each problem from all assignment sheets will be graded
- 35% of the final grade will be based on the assignments
- 65% of the final grade will be based on the final exam
- In the (written) final exam you will be asked to solve exercises similar to the assignments
- The final exam will take place at the end of the semester

Programming Assignments

- Presence problems need to be solved in the lab
- Further assignments are due on the following 5 days later at 10:00 in the morning, exact deadline displayed on the assignment sheet
- These assignments are also structured in a way that you can solve them during the lab session
- Solutions have to be submitted via the web interface of Grader to https://grader.eecs.jacobs-university.de
- Assignments are graded by the TAs
- Grading criteria written by the TAs: https://grader.eecs.jacobs-university.de/courses/350112/2018_1gB/Grading-Criteria-Python.pdf

Lab Sessions

- TAs will be available to help you in case of problems
- Optimize your time: solve the assignments during lab sessions
- Do not copy solutions, besides that it is not allowed, you will certainly fail the written final exam without practice in programming

Missing Homework, Quizzes, Exams according to AP

- https://www.jacobs-university.de/sites/default/files/bachelor_policies_v1.1.pdf (page 9)
- Illness must be documented with a sick certificate
- Sick certificates and documentation for personal emergencies must be submitted to the Student Records Office by the third calendar day
- Predated or backdated sick certificates will be accepted only when the visit to the physician precedes or follows the period of illness by no more than one calendar day
- Students must inform the Instructor of Record before the beginning of the examination or class/lab session that they will not be able to attend
- The day after the excuse ends, students must contact the Instructor of Record in order to clarify the make-up procedure
- Make-up examinations have to be taken and incomplete coursework has to be submitted by no later than the deadline for submitting incomplete coursework as published in the Academic Calendar

Syllabus

- Data collections
- Implementing basic data structures
- Creating tables using formatted output
- Generate HTML tables
- Object-oriented programming and design (classes, constructors)
- Using simple graphics (interactive graphics, mouse clicks, moving, textual input)
- Python's Standard Library
- Simple simulations

Agenda Week 1

- Reiterate data collections
- Implementing basic data structures
- Formatted output
- Creating tables using formatted output
- Generate HTML tables
- Object-oriented programming
  - Classes
  - Constructors
  - Methods (setters, getters, etc.)
  - Overloading operators

A `list` as a Stack

- On a stack the last element inserted is the first one retrieved (Last-In First-Out, LIFO)
- Common interface:
  - `push(element)` pushes an element onto the stack
  - `element = pop()` pops an element off the stack
  - very simple to support stack by lists and `append()` and `pop()`
  - `stack.py`

A `list` as a Queue

- The first element added to a queue is the first one retrieved (First-In First-Out, FIFO)
- Just think of a line at the cashier in your favorite supermarket
- A Python list is not really well-suited for queues
  - inserts and pops from the end of the list are fast
  - inserts or pops from the beginning are slow
- Import `collections.deque` to use queues
- `queue.py`

Tuples

- A tuple resembles a list, but is immutable
  - Indicate by enclosing its elements in ()

```
>>> fruits = ("apple", "banana")
>>> fruits
('apple', 'banana')
>>> meats = ("fish", "poultry")
>>> meats
('fish', 'poultry')
>>> food = meats + fruits
>>> food
('fish', 'poultry', 'apple', 'banana')
>>> veggies = ["celery", "beans"]
>>> tuple(veggies)
('celery', 'beans')
```

- Some of the operators and functions used with lists can be used in a similar fashion with tuples

Example: Finding the Mode of a List of Values

```
# Obtain the set of unique words and their
# frequencies, saving these associations in
# a dictionary
theDictionary = {}
for word in words:
    number = theDictionary.get(word, None)
    if number == None:
        # word entered for the first time
        theDictionary[word] = 1
    else:
        # word already seen, increment its number
        theDictionary[word] = number + 1

# Find the mode by obtaining the maximum value
# in the dictionary and determining its key
theMaximum = max(theDictionary.values())
for key in theDictionary:
    if theDictionary[key] == theMaximum:
        print("The mode is", key)
        break
```

Mapping

- Mapping applies a function to each value in a sequence and returns a new sequence of the results

```
>>> words = ["231", "20", "-45", "99"]
>>> map(int, words)           # Convert all strings to ints
<map object at 0x14cbd90>
>>> words                     # Original list is not changed
['231', '20', '-45', '99']
>>> words = list(map(int, words))   # Reset variable to change it
>>> words
[231, 20, -45, 99]
>>>
```

Filtering

- When filtering, a function called a predicate is applied to each value in a list
  - If predicate returns `True`, value is added to a new list; otherwise, value is dropped from consideration

```
>>> def odd(n): return n % 2 == 1

>>> list(filter(odd, range(10)))
[1, 3, 5, 7, 9]
>>>
```

Reducing

- When reducing, we take a list of values and repeatedly apply a function to accumulate a single data value

```
>>> from functools import reduce
>>> def add(x, y): return x + y

>>> def multiply(x, y): return x * y

>>> data = [1, 2, 3, 4]
>>> reduce(add, data)
10
>>> reduce(multiply, data)
24
>>>
```

Formatted Output

- The `format()` function allows finer control for formatting
- : followed by optional pair of characters
- Fill character (may not be })
- Alignment character
  - < left align, ^ center, > right align
- Minimal width integer
- Maximum width

General Format Specification

:	fill	align	sign	#	0	width	,	.precision	type
	Any character except }	< left > right ^ center = pad between sign and digits for numbers	+ force sign - sign if needed " " space or – as appropriate	prefix ints with 0b, 0o, or 0x	0-pad numbers	Minimum field width	use commas for grouping	Maximum field width for strings; number of decimal places for floats	`ints` b,c,d, n, o, x X; floats e, E, f, g, G, n, %

```
>>> "{0:,} {1:*>10,}".format(1234, 5678);
'1,234 *****5,678'
>>>
```

Formatting Output According to `locale`

```
>>> import locale
>>> x, y = (1234567890, 1234.56)
>>> locale.setlocale(locale.LC_ALL, "C")
'C'
>>> c = "{0:n} {1:n}".format(x, y)
>>> locale.setlocale(locale.LC_ALL, "en_US.UTF-8")
'en_US.UTF-8'
>>> en = "{0:n} {1:n}".format(x, y)
>>> locale.setlocale(locale.LC_ALL, "de_DE.UTF-8")
'de_DE.UTF-8'
>>> de = "{0:n} {1:n}".format(x, y)
>>> print(c)
1234567890 1234.56
>>> print(en)
1,234,567,890 1,234.56
>>> print(de)
1.234.567.890 1.234,56
>>>
```

Some Examples

- format.py
- How to choose the output:

```
1  import sys
2  x = int(input())
3  if (x == 1):
4    f = open("table.html", "w")
5  else:
6    f = sys.stdout
7  f.write("Hello")
8  f.close()
```

Modules Allow Code Reuse

- Up to now functions and the main program have been defined in same file
- Certain functions might be needed again and again and might be useful for different programs that are not related to each other
- Copying the code into several programs (and they will gradually diverge ..., a source for big trouble, your programs will behave differently in subtle ways ...) should not be done

Save Frequently Used Code into Modules

- Instead such code should be put into extra modules
- Python provides a simple way to create own modules
- Try to document your modules well; they might be reused even after a long time

Modules

```
1  """ Computer area and perimeter  of a circle
2  """
3  import math
4  def circle(radius):
5    area = math.pi * radius * radius
6    perimeter = 2 * math.pi * radius
7    # function returns two values!
8    return area, perimeter
9  r = 3
10 a, p = circle(r);
11 print("The area of a circle with radius {0:.2f}
       is {1:.2f}".format(r, a))
12 print("The perimeter of a circle with radius
       {0:.2f} is {1:.2f}".format(r, p))
```

Your Own Module (1)

Move the function into its own file, e.g., `mod_circle.py`

```
1  """    Circle module.
2         The module supports computations
3         regarding circles.
4  """
5  import math
6  def circle(radius):
7    """ computes area and perimeter of a circle
8        and returns result as tuple
9    """
10   area = math.pi * radius * radius
11   perimeter = 2 * math.pi * radius
12   return area, perimeter
13   # function returns two values!
```

Your Own Module (2)

Just import the file as module, it will be automatically interpreted

```
1 import mod_circle
2 r = 3
3 a, p = mod_circle.circle(r);
4 print("The area of a circle with radius {0:.2f}
      is {1:.2f}".format(r, a))
5 print("The perimeter of a circle with radius
      {0:.2f} is {1:.2f}".format(r, p))
```

Your Own Module (3)

```
Python 3.2.1 (default, Jul 18 2011, 16:24:40) [GCC] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> import mod_circle
>>> help(mod_circle)
Help on module mod_circle:

NAME
    mod_circle - Circle module.

DESCRIPTION
    The module supports computations regarding circles.

FUNCTIONS
    circle(radius)
        computes area and perimeter of circle and returns result as tuple

FILE
    /home/stamer/courses/350112/python/mod_circle.py


>>> help(mod_circle.circle)
Help on function circle in module mod_circle:

circle(radius)
    computes area and perimeter of circle and returns result as tuple

>>>
```

Your Own Module (4)

Alternatively you can explicitly import just the `circle()` function. This allows you to to call `circle()` without the module identifier.

```
1 from mod_circle import circle
2 r = 3
3 a, p = circle(r);
4 print("The area of a circle with radius {0:.2f}
      is {1:.2f}".format(r, a))
5 print("The perimeter of a circle with radius
      {0:.2f} is {1:.2f}".format(r, p))
```

Of course then you are unable to use a function with same name from different modules.
Using full qualifiers also allows you to keep track of the functions more easily.

HTML (Hypertext Markup Language) (1)

- Markup language describes general layout of the page
- Tags enclosed in angle brackets specify the structure and content of the page
- Tags are not shown, whitespace is always reduced to one

HTML (2)

```
<html>
  <head>
    <title>My webpage</title>
    <link rel="stylesheet" type="text/css" href=
   "style.css">
  </head>
  <body>
    <h1>My webpage</h1>
    This is my self-written webpage
  </body>
</html>
```

HTML Table

```
1  <table>
2    <tr>
3      <th>Name</th><th>Qty</th>
4    </tr>
5    <tr>
6      <td>apple</td><td>5</td>
7    </tr>
8    <tr>
9      <td>cherry</td><td>7</td>
10   </tr>
11 </table>
```

Name	Qty
apple	5
cherry	7

A Full Example HTML Page

```
1  <html>
2    <head>
3      <title>My webpage</title>
4    </head>
5    <body>
6      <h1>My table</h1>
7    <table>
8      <tr>  <th>Name</th><th>Qty</th> </tr>
9      <tr>  <td>apple</td><td>5</td>  </tr>
10     <tr>  <td>cherry</td><td>7</td> </tr>
11   </table>
12   </body>
13 </html>
```

style.css

```
1 th {
2   background-color: #999999;
3   min-width: 100px
4 }
5 td {
6   background-color: #DDDDDD;
7   min-width: 100px
8 }
```

style.css

How to Access your HTML File

- Assume you have stored the file to `/home/username/python/table.html`
- Then access the file with a browser via `file:///home/username/python/table.html`
- Note that there are three forward slashes (`///`)
- Or simply open the file with a browser

Getting Inside Objects and Classes

- Programmers who use objects and classes know:
  - Interface that can be used with a class
  - State of an object
  - How to instantiate a class to obtain an object
- Objects are abstractions
  - Package their state and methods in a single entity that can be referenced with a name
- Class definition is like a blueprint for each of the objects of that class

A First Example: The Student Class

- A course-management application needs to represent information about students in a course

```
>>> from student import Student
>>> s = Student("Maria", 5)
>>> print(s)
Name: Maria
Scores: 0 0 0 0 0
>>> s.setScore(1, 100)
>>> print(s)
Name: Maria
Scores: 100 0 0 0 0
>>> s.getHighScore()
100
>>> s.getAverage()
20
>>> s.getScore(1)
100
>>> s.getName()
'Maria'
>>>
```

The Student Class (1)

Student METHOD	WHAT IT DOES
`s = Student(name, number)`	Returns a **Student** object with the given **name** and **number** of scores. Each score is initially 0.
`s.getName()`	Returns the student's name.
`s.getScore(i)`	Returns the student's **i**th score. **i** must range from 1 through the number of scores.
`s.setScore(i, score)`	Resets the student's **i**th score to **score**. **i** must range from 1 through the number of scores.
`s.getAverage()`	Returns the student's average score.
`s.getHighScore()`	Returns the student's highest score.
`s.__str__()`	Same as **str(s)**. Returns a string representation of the student's information.

[TABLE 8.1] The interface of the **Student** class

The Student Class (2)

- Syntax of a simple class definition:

```
class <class name>(<parent class name>):
    <method definition-1>
    …
    <method definition-n>
```

  - The class name is a Python identifier
    - Typically capitalized
- Python classes are organized in a tree-like class hierarchy
  - At the top, or root of this tree is the `object` class
  - Some terminology: subclass, parent class
  - `student.py`

The Student Class (3)

```
def getAverage(self):
    """Returns the average score."""
    return sum(self._scores) / len(self._scores)

def getHighScore(self):
    """Returns the highest score."""
    return max(self._scores)

def __str__(self):
    """Returns the string representation of the student."""
    return "Name: " + self._name  + "\nScores: " + \
           " ".join(map(str, self._scores))
```

Docstrings

- Docstrings can appear at three levels:
  - Module
  - Just after class header
    - To describe its purpose
  - After each method header
    - Serve same role as they do for function definitions
- `help(Student)` prints the documentation for the class and all of its methods

Method Definitions

- Method definitions are indented below class header
- Syntax of method definitions similar to functions
  - Can have required and/or default arguments, return values, create/use temporary variables
  - Returns `None` when no `return` statement is used
- Each method definition must include a first parameter named `self`
- Example: `s.getScore(4)`
  - Binds the parameter `self` in the method `getScore` to the `Student` object referenced by the variable `s`

The `__init__` Method and Instance Variables

- Most classes include the `__init__` method

```
def __init__(self, name, number):
    """All scores are initially 0."""
    self._name = name
    self._scores = []
    for count in range(number):
        self._scores.append(0)
```

  - Class constructor
  - Runs automatically when user instantiates the class
- Example: `s = Student("Juan", 5)`
- Instance variables represent object attributes
  - Serve as storage for object state
  - Scope is the entire class definition

The __str__ Method

- Classes usually include an `__str__` method
  - Builds and returns a string representation of an object's state

```
def __str__(self):
    """Returns the string representation of the student."""
    return "Name: " + self._name  + "\nScores: " + \
           " ".join(map(str, self._scores))
```

- When the `str` function is called with an object, that object's `__str__` method is automatically invoked
- Perhaps the most important use of `__str__` is in debugging

Accessors and Mutators

- Methods that allow a user to observe but not change the state of an object are called accessors
- Methods that allow a user to modify an objects state are called mutators

```
def setScore(self, i, score):
    """Resets the ith score, counting from 1."""
    self._scores[i - 1] = score
```

- Tip: if there is no need to modify an attribute (e.g., a student's name), do not include a method to do that

The Lifetime of Objects

- The lifetime of an object's instance variables is the lifetime of that object
- An object becomes a candidate for the 'graveyard' when it can no longer be referenced

```
>>> s = Student("Sam", 10)
>>> csci111 = [s]
>>> csci111
[<__main__.Student instance at 0x11ba2b0>]
>>> s
<__main__.Student instance at 0x11ba2b0>
>>>
>>> s = None
>>> csci111.pop()
<__main__.Student instance at 0x11ba2b0>
>>> print s
None
>>> csci111
[]
```

`Student` object still exists, but interpreter will recycle its storage during **garbage collection**

Visibility of Class Components (1)

- public: all member variables and methods are public by default in Python
- protected: accessible only from within the class and it's subclasses, in Python by prefixing the name of your member with a single underscore
- private: nobody should be able to access it from outside the class, in Python by prefixing with at least two underscores and suffixing with at most one underscore

Visibility of Class Components (2)

```
1  class Cup:
2      def __init__(self, mtype, content, color="Green"):
3      # default value for parameter
4          self.mtype = mtype              # public variable
5          self.__content = content        # private variable
6          self._color = color             # protected variable
7
8      def fill(self, beverage):
9          self.__content = beverage
10
11     def empty(self):
12         self.__content = None
13
14 cupobj1 = Cup("Paper", "Water")
15 print(cupobj1.mtype, cupobj1._color)
16 cupobj2 = Cup("Glas", None, "Red")
17 cupobj2.fill("Lemonde")
18 print(cupobj2.mtype, cupobj2._color)
19 print(cupobj2.__content) # does not work because private
```

Rules of Thumb for Defining a Simple Class

- Before writing a line of code, think about the behavior and attributes of the objects of new class
- Choose an appropriate class name and develop a short list of the methods available to users
- Write a short script that appears to use the new class in an appropriate way
- Choose appropriate data structures for attributes
- Fill in class template with `__init__` and `__str__`
- Complete and test remaining methods incrementally
- Document your code

How to Play Craps

- You roll two dice
- If the sum is one of 2, 3, 12 you lose
- If the sum is one of 7 or 11 you win
- Otherwise you continue rolling
  - If the sum is 7 you lose
  - If you get the same sum as in your initial roll you win
  - Otherwise you continue rolling

Case Study: Playing the Game of Craps

- Request:
  - Write a program that allows the user to play and study the game of craps
- Analysis: define `Player` and `Die` classes
  - User interface: prompt for number of games to play

```
>>> playOneGame()
(2, 2) 4
(2, 1) 3
(4, 6) 10
(6, 5) 11
(4, 1) 5
(5, 6) 11
(3, 5) 8
(3, 1) 4

You win!
```

```
>>> playManyGames()
Enter the number of games: 100
The total number of wins is 49
The total number of losses is 51
The average number of rolls per win is 3.37
The average number of rolls per loss is 4.20
The winning percentage is 0.490
```

Case Study: Design

`Player` METHOD	WHAT IT DOES
`p = Player()`	Returns a new player object.
`p.play()`	Plays the game and returns `True` if there is a win, `False` otherwise.
`p.getNumberOfRolls()`	Returns the number of rolls.
`p.__str__()`	Same as `str(p)`. Returns a formatted string representation of the rolls.
`Die` METHOD	**WHAT IT DOES**
`d = Die()`	Returns a new die object whose initial value is 1.
`d.roll()`	Resets the die's value to a random number between 1 and 6.
`d.getValue()`	Returns the die's value.
`d.__str__()`	Same as `str(d)`. Returns the string representation of the die's value.

[TABLE 8.2] The interfaces of the `Die` and `Player` classes

Case Study: Implementation (1)

```
"""
File: die.py

This module defines the Die class.
"""

from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        """The initial face of the die."""
        self._value = 1

    def roll(self):
        """Resets the die's value to a random number
        between 1 and 6."""
        self._value = randint(1, 6)

    def getValue(self):
        return self._value

    def __str__(self):
        return str(self._value)
```

Case Study: Implementation (2)

```
"""
File: craps.py

This module studies and plays the game of craps.
"""

from die import Die

class Player(object):

    def __init__(self):
        """Has a pair of dice and an empty rolls list."""
        self._die1 = Die()
        self._die2 = Die()
        self._rolls = []

    def __str__(self):
        """Returns the string rep of the history of rolls."""
        result = ""
        for (v1, v2) in self._rolls:
            result = result + str((v1, v2)) + " " + \
                     str(v1 + v2) + "\n"
        return result

    def getNumberOfRolls(self):
        """Returns the number of the rolls in one game."""
        return len(self._rolls)
```

Data-Modeling Examples

- As you have seen, objects and classes are useful for modeling objects in the real world
- In this section, we explore several other examples

Rational Numbers

- A rational number consists of two integer parts, a numerator and a denominator
  - Examples: $1/2$, $-2/3$, $14/123$, etc.
- Python has no built-in type for rational numbers
  - We will build a new class named `Rational`

```
>>> oneHalf = Rational(1, 2)
>>> oneSixth = Rational(1, 6)
>>> print(oneHalf)
1/2
>>> print(oneHalf + oneSixth)
2/3
>>> oneHalf == oneSixth
False
>>> oneHalf > oneSixth
True
```

Operators need to be **overloaded**

Rational Number Arithmetic and Operator Overloading (1)

OPERATOR	METHOD NAME
+	`__add__`
-	`__sub__`
*	`__mul__`
/	`__div__`
%	`__mod__`

[TABLE 8.3] Built-in arithmetic operators and their corresponding methods

- Object on which the method is called corresponds to the left operand
- For example, the code `x + y` is actually shorthand for the code `x.__add__(y)`

Rational Number Arithmetic and Operator Overloading (2)

- To overload an arithmetic operator, you define a new method using the appropriate method name
- Code for each method applies a rule of rational number arithmetic

TYPE OF OPERATION	RULE
Addition	$n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1) / d_1d_2$
Subtraction	$n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1) / d_1d_2$
Mutiplication	$n_1/d_1 * n_2/d_2 = n_1n_2 / d_1d_2$
Division	$n_1/d_1 / n_2/d_2 = n_1d_2 / d_1n_2$

[TABLE 8.4] Rules for rational number arithmetic

Rational Number Arithmetic and Operator Overloading (3)

```
def __add__(self, other):
    """Returns the sum of the numbers."""
    #Self is the left operand and other is the right operand
    newNumer = self._numer * other._denom + \
               other._numer * self._denom
    newDenom = self._denom * other._denom
    return Rational(newNumer, newDenom)
```

- Operator overloading is another example of an abstraction mechanism
- We can use operators with single, standard meanings even though the underlying operations vary from data type to data type

Comparison Methods

OPERATOR	MEANING	METHOD
==	Equals	__eq__
!=	Not equals	__ne__
<	Less than	__lt__
<=	Less than or equal	__le__
>	Greater than	__gt__
>=	Greater than or equal	__ge__

[TABLE 8.5] The comparison operators and methods

Equality and the __eq__ Method

- Not all objects are comparable using < or >, but any two objects can be compared for == or !=

  `twoThirds < "hi there"` should generate an error

  `twoThirds != "hi there"` should return `True`

- Include `__eq__` in any class where a comparison for equality uses a criterion other than object identity

```
def __eq__(self, other):
    """Tests self and other for equality."""
    if self is other:                    # Object identity?
        return True
    elif type(self) != type(other):      # Types match?
        return False
    else:
        return self._numer == other._numer and \
               self._denom == other._denom
```

Savings Accounts and Class Variables (1)

SavingsAccount METHOD	WHAT IT DOES
`a = SavingsAccount(name, pin, balance = 0.0)`	Returns a new account with the given name, PIN, and balance.
`a.deposit(amount)`	Deposits the given amount from the account's balance.
`a.withdraw(amount)`	Withdraws the given amount from the account's balance.
`a.getBalance()`	Returns the account's balance.
`a.getName()`	Returns the account's name.
`a.getPin()`	Returns the account's PIN.
`a.computeInterest()`	Computes the account's interest and deposits it.
`__str__(a)`	Same as **`str(a)`**. Returns the string representation of the account.

[TABLE 8.5] The interface for **SavingsAccount**

Savings Accounts and Class Variables (2)

```
class SavingsAccount(object):
    """This class represents a Savings account
    with the owner's name, PIN, and balance."""

    RATE = 0.02

    def __init__(self, name, pin, balance = 0.0):
        self._name = name
        self._pin = pin
        self._balance = balance

    def __str__(self):
        result =  'Name:    ' + self._name + '\n'
        result += 'PIN:     ' + self._pin + '\n'
        result += 'Balance: ' + str(self._balance)
        return result

    def getBalance(self):
        return self._balance

    def getName(self):
        return self._name

    def getPin(self):
        return self._pin
```

Savings Accounts and Class Variables (3)

```
def deposit(self, amount):
    """Deposits the given amount and returns the
    new balance."""
    self._balance += amount
    return self._balance

def withdraw(self, amount):
    """Withdraws the given amount.
    Returns None if successful, or an
    error message if unsuccessful."""
    if amount < 0:
        return 'Amount must be >= 0'
    elif self._balance < amount:
        return 'Insufficient funds'
    else:
        self._balance -= amount
        return None

def computeInterest(self):
    """Computes, deposits, and returns the interest."""
    interest = self._balance * SavingsAccount.RATE
    self.deposit(interest)
    return interest
```

Putting the Accounts into a Bank (1)

```
>>> from bank import Bank, SavingsAccount
>>> bank = Bank()
>>> bank.add(SavingsAccount("Wilma", "1001", 4000.00))
>>> bank.add(SavingsAccount("Fred", "1002", 1000.00))
>>> print(bank)
Name:    Fred
PIN:     1002
Balance: 1000.00
Name:    Wilma
PIN:     1001
Balance: 4000.00
>>> account = bank.get("1000")
>>> print(account)
None
>>> account = bank.get("1001")
>>> print(account)
Name:    Wilma
PIN:     1001
Balance: 4000.00
>>> account.deposit(25.00)
4025
>>> print(account)
Name:    Wilma
PIN:     1001
Balance: 4025.00
>>> print(bank)
```

Putting the Accounts into a Bank (2)

Bank METHOD	WHAT IT DOES
`b = Bank()`	Returns a bank.
`b.add(account)`	Adds the given account to the bank.
`b.remove(pin)`	Removes the account with the given PIN from the bank and returns the account. If the pin is not in the bank, returns **None**.
`b.get(pin)`	Returns the account associated with the PIN if the PIN is in the bank. Otherwise, returns **None**.
`b.computeInterest()`	Computes the interest on each account, deposits it in that account, and returns the total interest.
`__str__(b)`	Same as **str(b)**. Returns a string representation of the bank (all the accounts).

[TABLE 8.6] The interface for the **Bank** class

Putting the Accounts into a Bank (3)

```
class Bank(object):

    def __init__(self):
        self._accounts = {}

    def __str__(self):
        """Return the string rep of the entire bank."""
        return '\n'.join(map(str, self._accounts.values()))

    def add(self, account):
        """Inserts an account using its PIN as a key."""
        self._accounts[account.getPin()] = account

    def remove(self, pin):
        return self._accounts.pop(pin, None)

    def get(self, pin):
        return self._accounts.get(pin, None)

    def computeInterest(self):
        """Computes interest for each account and
        returns the total."""
        total = 0.0
        for account in self._accounts.values():
            total += account.computeInterest()
        return total
```

Summary (1)

- A simple class definition consists of a header and a set of method definitions
- In addition to methods, a class can also include instance variables
- Constructor or `__init__` method is called when a class is instantiated
- A method contains a header and a body
- An instance variable is introduced and referenced like any other variable, but is always prefixed with `self`

Summary (2)

- Some standard operators can be overloaded for use with new classes of objects
- When a program can no longer reference an object, it is considered dead and its storage is recycled by the garbage collector
- A class variable is a name for a value that all instances of a class share in common