Object-oriented programming with Python

CPEN333 - 2023 W1

University of British Columbia

Introduction

- Object-oriented programming enables us to develop large-scale software effectively.

- In the procedural paradigm (like in the programming language C), we focus on designing functions.

- With the object-oriented thinking, software design focuses on objects.
  - An object couples related data and the required operations together.

- In this set of slides, we first examine object-oriented programming using Python and then we discuss useful data types, exception handling, …

Objectives

- In this set of slides, we continue discussing python's fundamental concepts, by learning object oriented programming and more concepts.

- By the end of this set of slides, you will be able to:
  - Design classes and define objects
  - Use python data types, e.g. list and string
  - Implement exception handling
  - Use python standard library modules
  - …

Note

- The Python programming language is a tool that we use to develop software to solve problems.

- The programming language is not the ultimate goal, but an important skill to achieve it.

What is a Data Type?

A type is a set of values and the operations that are permitted on the values.

- Essentially, a type is a way to describe a data item more precisely by articulating not only the values that the item can take but also the operations that are permitted on the item.

- Python comes with many built-in types.
  - And we can import or create new types.

Object-oriented programming

- As an object-oriented programming language, objects are python's abstraction for data.
- An object can contain other objects.
- The Python's versatility allows us to use either of or a mix or procedural and object-oriented paradigm.

Class

- We use a class to define the properties and behaviours of a type or related object.

- A class is a blueprint (or a template or a contract) for creating an object.
  - We can use a class to create many objects.

- The coupling of related data (attributes or properties) and methods (behaviour) is fundamentally important.

Data Type Using Class

- A type's attributes (or properties or states or values or data members) are represented by specific variables called data fields.

- A type's behaviour (or operations or actions or function members) are represented by its methods.

Note: data fields have a very special meaning and purpose. We should not confuse them with other variables such as local variable, class variables, …

Class definition

- The following syntax is used to define a class:

```
class ClassName:
    initializer
    methods
```

- Objects are created from classes.

- Instance methods are the function members of the class.
  - They are used to define the intended behaviour.
  - The first parameter for all instance methods is self. It is used in the definition but not when the method is called.

- Note that the way data fields are created/set in Python is different from C-based OO languages such as C#.

Initializer

- The initializer is a special method that is automatically invoked to initialize a new object's state when an object is created.
  - It is somewhat similar to the *constructor method* in other languages like C#.
- Initializers can perform any action, but they are designed to perform initializing actions, such as creating an object's data fields with initial values.
- To create an initializer in Python, we use the __init__ () method in a class (note that *init* is preceded and followed by two underscores).
  - An example of a simple initializer for a class c:

```
class Circle:
    def __init__(self, radius = 1):    # initializer
        self.radius = radius            # data field
```

Instantiation

- An object is an instance of a class.
- Once a class is defined, we can create objects from the class with a constructor.
  - To construct an object of a given class, we call the class with appropriate arguments. The Syntax for a constructor is:

    `ClassName (arguments)`
- We can create as many instances of a class as needed.
  - Note that the *self* parameter in the __init__ method is automatically set to reference the object that is just created and must not be included in the call.
  - Examples:

```
circle1 = Circle(5)    # creating a circle object with radius 5

circle2 = Circle(2)    # creating another circle object with radius 2

circle3 = Circle()     # creating another circle object with default radius 1
```

- Creating an instance of a class is referred to as instantiation. The terms object and instance are often used interchangeably.

Example: Circle

- Assume that in our program we want to work with circles.
- During the design process, we first decide what attributes and operations we need.
  - Considered data field: its radius
  - Considered operations: get perimeter, get area

```
import math

class Circle:                                    # class name
    def __init__(self, radius = 1):              # initializer (i.e. constructor method)
        self.radius = radius                     # data field
    def getPerimeter(self):                      # method for getting perimeter
        return 2 * self.radius * math.pi
    def gerArea(self):                           # method for getting area
        return self.radius * self.radius * math.pi

circle = Circle(5)            # creating/constructing a circle object with radius 5
print(circle.getPerimeter()) # calling its getPerimeter method
```

class definition

Another example

- Here is an example of a class and its methods:

```
class Dog:
    kind = 'canine' # class variable shared by all instances

    def __init__(self, name): # the constructor method
        self.name = name  # instance variable unique to each instance
        self.tricks = []  # creates a new empty list for each dog

    def add_trick(self, trick): # a class method
        self.tricks.append(trick)
```

- Note that:
  - the initializer method is __init__
  - an example of creating a Dog object is `fido = Dog("Fido")`
  - an example of calling a method is `fido.add_trick("roll over")`

Objects (or Instances)

- In Python, all data (including numbers and strings) are objects.
  - https://docs.python.org/3/reference/datamodel.html#objects
- Every object has an identity, a type and a value.
- The object's identity never changes once it is created.
  - We can use the function `id()` to get the identity of an object.
  - In CPython (the python implementation we use), `id(x)` is the memory address where x is stored.
- The object type determines the data type and operations it supports.
  - We can use the function `type()` to get the type of an object.
  - Similar to an object's identity, its type is unchangeable.

Examples

- Example of an `int` object:

```
>>> x: int = 2
>>> id(x)
2180568082768
>>> type(x)
<class 'int'>
```

- Example of a `str` object:

```
>>> y = "hello"
>>> id(y)
2180608953200
>>> type(y)
<class 'str'>
```

Mutable vs immutable

- Objects whose value is unchangeable once created are called immutable.
  - Example: numbers, strings and tuples are immutable
    - The string “Hello World” is an example of an immutable object.
- Objects whose value can change are called mutable
  - Example: dictionaries and lists are mutable
- Immutability helps with writing robust software.

More on the self parameter

- self is a parameter (by convention) that references the object itself.
- It is used in the implementation of an instance method (its first parameter) or defining a data field (using dot notation).
- We use self to access class members in a class definition.
- The scope of an instance variable is the entire class once created.

Note that self is not used when the method is called.

```
class Example:
    def __init__(self, ...): # first param
        self.x = 1  # create/modify x
        ...
    def method1 (self, ...):
        self.y = 2.4 # create/modify y
        ...
    def method2 (self, ...):
        ...
        self.method1(...) # invoking m1
        ...
```

Hiding data fields

- Making data fields private (only available within the class) protects data and makes the class easier to maintain. There are techniques to make a data field private in python.
- Using a single _ (underscore) at the beginning of a data field identifier indicates that it is meant to be a private field.
  - This, however, does NOT prevent the data being accessed and modified.
- Alternatively, we can use __ (two underscores) at the beginning of a data field. This uses the python's *name mangling* to provide data hiding.
- Example:

```
def __init__(self, radius = 1):
    self._radius = radius
```

```
def __init__(self, radius = 1):
    self.__radius = radius
```

The str class

- Python's own str class is a good example of class design.
  - https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
- Creating strings

using the constructor

```
s1 = str()
s2 = str("hello")
```

alternatively

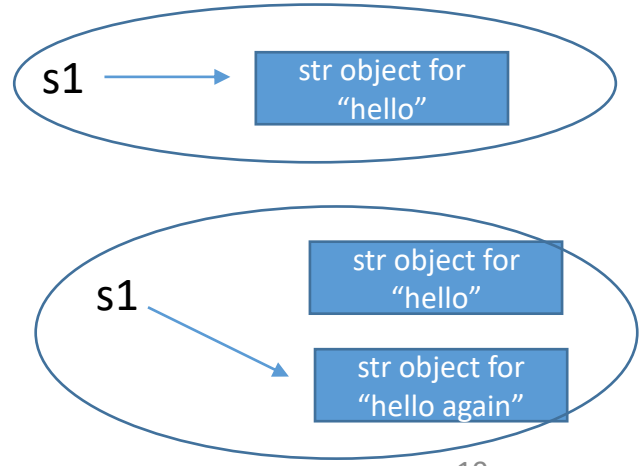
simplified notation

```
s1 = ""
s2 = "hello"
```

- A str object is immutable (its content cannot be changed once created).

```
>>> s1 = "hello"
>>> print("The s1's id is: ", id(s1))
The s1's id is:  1385520247856

>>> s1 = "hello again"
>>> print("After variable reassignment, s1's id is: ", id(s1))
After variable reassignment, s1's id is:  1385520238768
```



The str class methods

- There are many useful methods.
  - https://docs.python.org/3/library/stdtypes.html?highlight=str#string-methods

- Some examples:

```
msg = "Let's be hopeful"
print(msg.islower()) #at least one char and all lower case?
print(msg.isdigit()) #all number character?
print("2022".isdigit())
s2 = msg.upper()
print(s2)
s3 = s2.replace("BE", "STAY") #replace all BE with STAY
print(s3)
```

output

```
False
False
True

LET'S BE HOPEFUL

LET'S STAY HOPEFUL
```

Other functions and operators

- Several Python's built-in functions can be used with strings (as with many other types).

```
msg = "welcome"
print(len(msg)) # legnth of msg
print(max(s))   # largest char in msg
```

output

```
7
w
```

- Similar to C or C#, a character in a string can be accessed using the index operator ([]).
  - Python also has a slicing operator `[start:end]`, that gives a substring from index `start` to index `end-1`.

```
msg = "Hello again"
print(msg[0])   # index operator
print(msg[0:5]) # slicing operator
```

output

```
H
Hello
```

  - Since strings are immutable, an operation like the one below is illegal.

```
msg = "Hello again"
msg[0] = 'h' # illegal --> will result in a TypeError exception
```

Built-in Types

- Python comes with a good set of built-in types.

- The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions.

- Examples of sequence types are str, list, tuple
  - https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range
  - We have already discussed the str type.
  - Tuples are like lists, except that lists are mutable while tuples are immutable (their elements are fixed and cannot be changed after it is created).
    - See: https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

Built-in Types (cont.)

- An example of set types is `set` (like lists, but nonduplicate and no order)

- An example of mapping types is `dict` (dictionary: a collection of key/value pairs)

- Pythons does not have a built-in array type, instead for example we can use lists.
  - There is though an array type used in libraries such as numpy.

List

- A list can store a collection of data of any size.
  - You can access any list element by using the index operator, [].
  - You can use list methods to manipulate lists.
  - Lists are mutable.
  - Example:

```
list1 = [] # create an empty list

list2 = [2,3,4] # create a list with elements 2, 3, 4

list3 = ["red","green"] # create a list with strings

list4 = [2,"three",4] # A list can contain mixed types

two = list4[0] # stores 2 (element 0 of list4) in two

list2.append(5) # modifies list2 to have 2, 3, 4, 5
```

Set

- Sets are sequences like lists, but the elements in a set are nonduplicate and are not placed in any particular order.
  - A set can have elements of the same type or mixed types.
  - Sets are mutable.
  - https://python-reference.readthedocs.io/en/latest/docs/sets/
  - Examples:

```
set1 = set() # create an empty set

set2 = {2,3,4} # create a set with elements 2, 3, 4

set3 = {2,"three",4} # A set can contain mixed types

set2.add(5) # modifies set2 to have 2, 3, 4, 5

set3.remove(2) # removes 2 from set3
```

Dictionary

- A dictionary (map) is a container object that stores a collection of key-value pairs.
  - The elements in the dictionary is in the form *key:value*.
  - A dictionary cannot contain duplicate keys (since the keys are used for indexing).
  - https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

```
students = {} # create an empty dictionary

students[111] = "John" # adds 111:"john" to students
students[112] = "Mary" # adds 112:"Mary" to students
students[112] = "Ann"  # changes 112:"Mary" to 112:"Ann"
print(students)        # prints: {111: 'John', 112: 'Ann'}
```

Exception Handling

- Exception handling enables a program to deal with exceptions and potentially continue its normal execution.
  - An exception is an error that occurs at runtime.
  - Example:

```
>>> x = int(input("Enter x to calculate 1/x: "))
Enter x to calculate 1/x: 0
>>> 10/x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

- We can wrap the code that might raise (or throw) an exception in a try clause of the python's exception handling syntax.
  - https://docs.python.org/3.10/tutorial/errors.html

Exception Handling (cont.)

- The syntax is:

```
try:
    #body here
except ExceptionTypeHere:
    #handler here
```

- or more general syntax:

```
try:
    #body here
except ExceptionType1:
    #handler for ExceptionType1
except ExceptionType2:
    #handler for ExceptionType2
...
except: #executed if exception doesn't match previous types
    #handler for except
else: #executed if no exception is raised
    #handler for else
finally: #executed always (e.g. for cleanup)
    #handler for finally
```

Examples

- Example1:

```
try:
    x = int(input("Enter an integer: "))
except ValueError:
    print("Oops, that is not a valid number.")
```

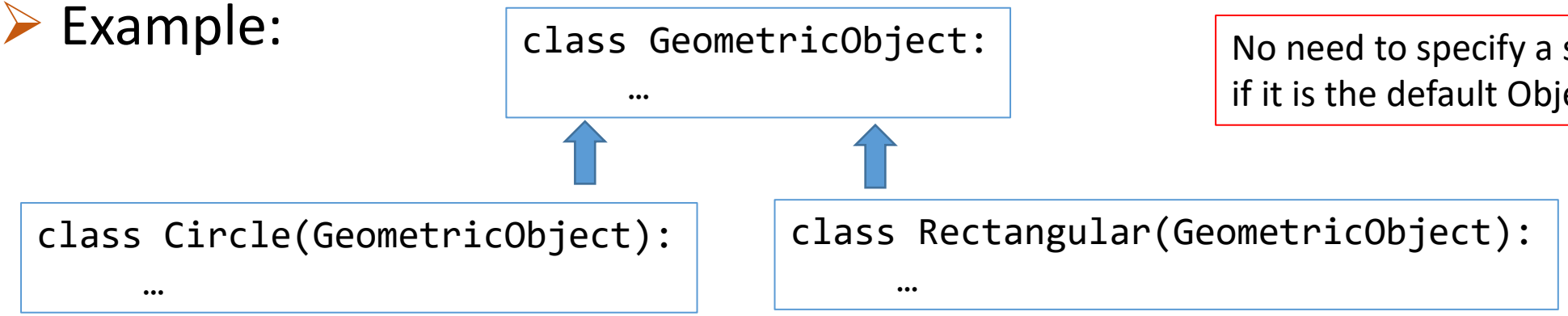
- Example2:

```
>>> try:
...     x = int(input("Enter x to calculate 1/x: "))
...     print(1/x)
... except ZeroDivisionError:
...     print("x cannot be 0")
... except:
...     print("Something is wrong with the input")
...
Enter x to calculate 1/x: 0
x cannot be 0
```

- For a list of built-in exceptions see: https://docs.python.org/3.10/library/exceptions.html

Inheritance

- Object oriented programming allows us to define new classes from existing ones. This is called inheritance.
- Inheritance allows us to define a more general class (superclass or parent) and later extend it to more specialized classes (subclass or derived).
- Every class in Python is descended from the *object* class.
- We use the following syntax: `class subclass(superclass):`
- Example:



No need to specify a superclass, if it is the default Object class.

Python standard library

- Python's standard library is very extensive, offering a wide range of facilities that is distributed with Python.
  - https://docs.python.org/3/library/
- Examples are:
  - Data types
  - Text processing services
  - Generic operating system services
  - concurrent execution
  - Graphical user interface
  - Internet protocols and support
  - Networking and interprocess communications
  - …

References

- There is a great amount of good information on Python on the Internet.
  - Python documentation: https://docs.python.org/3/
    - https://docs.python.org/3/tutorial/classes.html
  - Standard library: https://docs.python.org/3/library/
  - https://www.w3schools.com/python/
  - …
- There are also many good books, e.g.:
  - Daniel Liang's ***Intro to Programming using Python***