COMP 1023 Introduction to Python Programming
Introduction to Object-Oriented Programming

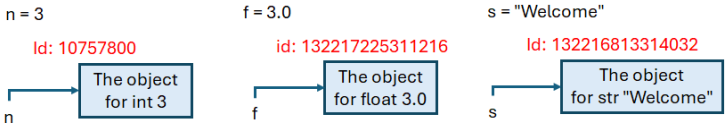
Dr. Cecia Chan, Prof. SC Cheung, Dr. Alex Lam, Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Introduction

- In Python, everything is an object, including numbers, strings, and other data types.
  - Objects have an identity, type, and value.
  - The `id()` function returns the unique identifier of an object.
  - The `type()` function returns the type of an object.

```
n = 3
print(id(n))      # Print 10757800
print(type(n))    # Print <class 'int'>
f = 3.0
print(id(f))      # Print 132217225311216
print(type(f))    # Print <class 'float'>
s = "Welcome"
print(id(s))      # Print 132216813314032
print(type(s))    # Print <class 'str'>
```



Object-Oriented Programming

- Object-Oriented Programming (OOP) involves the use of objects to create programs.
- An object represents an entity in the real world that can be distinctly identified.
- Examples of objects: a student, a desk, a circle.
- Key Characteristics of Objects:
  - Identity:
    - Python assigns each object a unique ID for identifying the object at runtime.
  - States (Attributes):
    - Represented by variables.
    - Example: radius of a circle object.
  - Behaviors (Methods):
    - Actions performed by the object through methods (functions).
    - Example: `getArea()` for circle objects.
- A Python class serves as a template, a blueprint, a contract for creating objects, defining their data fields and methods.

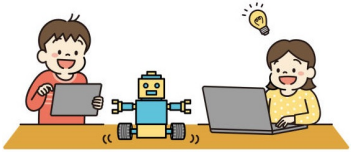
Class and Object

- A class is a user-defined data type from which objects are created.
- A class provides a means of bundling data (instance variables) and functionality (methods) together.
  - Instance Variables:
    - Variables that belong to an object.
    - Public by default and can be accessed using the dot (.) operator.
  - Instance Methods:
    - Defined with an extra first parameter, `self`.
    - No value is given for `self` when calling the method; Python provides it automatically.
- Instance variables and methods are accessed using the object.
- The `__init__` method is an initializer.
  - Used to initialize the instance variables of objects.
- Objects are instances of a class.

Defining Classes

Syntax

```
class <class-name>:
  # __init__ is an initializer
  def __init__(self, <argument-list1>):
      self.<instance-variable-name1> = <value1>
      self.<instance-variable-name2> = <value2>
      ...
  def <method1-name>(self, <argument-list2>):
      <method1-statements>
  def <method2-name>(self, <argument-list3>):
      <method2-statements>
  ...
```



Parameters

- `<class-name>`: The name of the class
- `<method1-name>`, `<method2-name>`, ...: Instance method names
- `<argument-list1>`, `<argument-list2>`, `<argument-list3>`, ...: Instance method parameters (i.e., variables)
- `<instance-variable-name1>`, `<instance-variable-name2>`, ...: Instance variable names
- `<value1>`, `<value2>`, ...: Value assigned to the instance variables
- `<method1-statements>`, `<method2-statements>`, ...: Python statements

Creating Objects, Calling a Method, and Modifying an Instance Variable

- An object is created in memory using a constructor for the class.
- Then, the class's `__init__` method is called to initialize the object.

Note: In Python, a constructor is created using the `__new__` method. (We won't be discussing this in this course!)

Syntax

```
<object-name> = <class-name>(<arguments>) # Create an object with a constructor
<object-name>.<instance-method-name>(<arguments>)  # Call a method
<object-name>.<instance-variable-name> = <value>   # Modify an instance variable
```

Parameters:

- `<object-name>`: The name of the object.
- `<class-name>`: The name of the class.
- `<arguments>`: The values passed to the constructor or method.
- `<instance-method-name>`: The name of the instance method.
- `<instance-variable-name>`: The name of the instance variable.
- `<value>`: The value assigned to the instance variable.

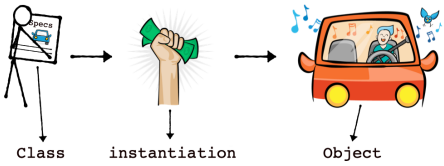
Object Instantiation

1. The constructor creates an object in the memory for the class

Object
Instance variables:

2. Then, the class's __init__ method is called to initialize the object. The self parameter in the __init__ method references the object that was just created

__init__(self, ...)



Example

```
# Filename: circle.py
import math                                 # Import math library

class Circle:                               # Define a new type Circle
    def __init__(self, radius):             # Define an initializer with parameter: radius
        self.radius = radius                # Define an instance variable radius
                                            # and assign it with parameter: radius
    def area(self):                         # Define the instance method: area
        return math.pi * self.radius ** 2   # Compute and return the area of the circle

    def circumference(self):                # Define the instance method: circumference
        return 2 * math.pi * self.radius    # Compute and return the circumference of
                                            # the circle
def main():
    circleObj = Circle(10)                  # Define an object of Circle named circleObj
    circleObj.radius = 100                  # Modify circleObj's radius to 100
    print("Area:", circleObj.area())                    # Call area()
    print("Circumference:", circleObj.circumference())  # Call circumference()

if __name__ == "__main__":
    main()
```

Output:

```
Area: 31415.926535897932
Circumference: 628.3185307179587
```

self Parameter

- self is a parameter that references the object itself.
- When a method is called on an object, self is set to that object.
- Using self, you can access the object's members (attributes and methods) within a class definition.

Scope of Instance Variables

```
def ClassName:
    def __init__(self, ...):
        self.x = 1      # Create/modify x
        ...

    def m1(self, ...):
        self.y = 2      # Create/modify y
        ...
        z = 5           # Create/modify z
        ...

    def m2(self, ...):
        self.y = 3      # Create/modify y
        ...
        u = self.x + 1  # Create/modify u
        self.m1(...)    # Invoke m1
```

Scope of self.x and self.y is the entire class

Scope of z

Copying Objects?

```python
# Filename: circle_shallow_copy.py
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

    def circumference(self):
        return 2 * math.pi * self.radius

def main():
    circleObj1 = Circle(10)
    circleObj2 = Circle(20)
    circleObj1 = circleObj2

if __name__ == "__main__":
    main()
```

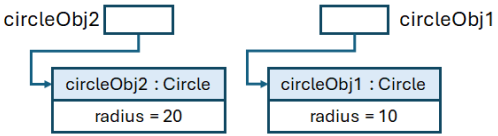
- `circleObj1` and `circleObj2` are two `Circle` objects.
- When you execute:

  `circleObj1 = circleObj2`

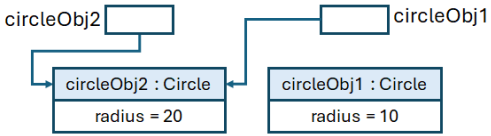
- This copies the reference of `circleObj2` to `circleObj1`, not the contents.

Copying Objects?

Before executing circleObj1 = circleObj2



After executing circleObj1 = circleObj2



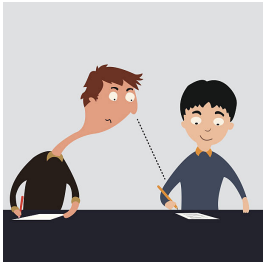
This object is not used.
It is now garbage!

Copying Objects!!!

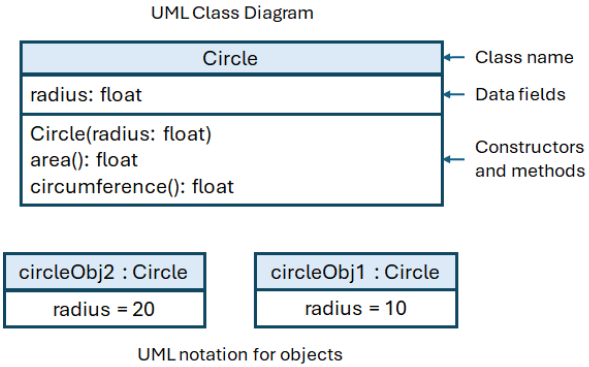
```python
# Filename: circle_deep_copy.py
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

    def circumference(self):
        return 2 * math.pi * self.radius

def main():
    circleObj1 = Circle(10)
    circleObj2 = Circle(20)
    circleObj1.radius = circleObj2.radius

    print("Circle 1 radius:", circleObj1.radius)
    print("Circle 1 area:", circleObj1.area())
    print("Circle 1 circumference:", circleObj1.circumference())
```

Output:

```
Circle 1 radius: 20
Circle 1 area: 1256.6370614359173
Circle 1 circumference: 125.66370614359172
```



UML (Unified Modeling Language) Diagram



- Data field: `dataFieldName: dataFieldType`
- Constructors: `ClassName(parameterName: parameterType)`
- Methods: `methodName(parameterName: parameterType) : returnType`
- The `__init__` method does not need to be listed in the UML diagram.

Public, Private and Protected Members

- As mentioned, all members in a Python class are public by default, i.e., any member can be accessed from outside the class. To restrict access to the members, you can make them protected or private.
- Protected members of a class are accessible from within the class and also available to its subclasses (will not be covered in this course). By convention, Python makes an instance variable/method protected by adding a prefix _ (single underscore) to it.
- Private members of a class are accessible only from within the class. By convention, Python makes an instance variable/method private by adding a prefix __ (double underscore) to it.

Note

- "By convention" means the responsible programmer should refrain from accessing and modifying instance variables prefixed with _ or __ from outside their class. However, if they do so, it will still work. :(
- Modern IDE, especially VS Code and PyCharm, yields a warning when you do so.

Name Mangling

- Name mangling in Python is a mechanism that the interpreter uses to modify the names of class attributes that begin with double underscores (`__`).
- When an attribute name starts with two underscores (but does not end with two underscores):
  - Python internally changes the name by adding a single underscore and the class name as a prefix.
  - For example, an attribute named `__myattribute` in a class named `MyClass` becomes `_MyClass__myattribute`.
- This mangling is done at the time of object creation.
- The mangled name is used internally by the interpreter.
- Name mangling does not provide true privacy.

Name mangling is not a security feature; it is more of a convention to avoid naming conflicts.

Example

```python
# Filename: person.py
class Person:
    def __init__(self, name="Tom",
                       age=18,
                       gender='M'):
        # private instance variable
        self.__name = name
        # private instance variable
        self.__age = age
        # private instance variable
        self.__gender = gender

    def print_info(self):
        print("--- Print Person ---")
        print("Name: " + self.__name)
        print("Age: " + str(self.__age))
        print("Gender: " + self.__gender)
```

```python
def main():
    desmond = Person("Haha", 18, "M")

    # print(f"Name: {desmond.__name}")      # Error
    # print(f"Age: {desmond.__age}")        # Error
    # print(f"Gender: {desmond.__gender}") # Error
    # Okay, but ...
    desmond.__name = "Desmond"
    # Okay, but ...
    desmond.__age = 19
    desmond.print_info()

if __name__ == "__main__":
    main()
```

How to retrieve and modify instance variable values? You need to define accessors and mutators!

Code after Name Mangling

```python
class Person:
    def __init__(self, name="Tom",
                       age=18,
                       gender='M'):
        self._Person__name = name
        self._Person__age = age
        self._Person__gender = gender

    def print_info(self):
        print("--- Print Person ---")
        print("Name: " +
              self._Person__name)
        print("Age: " +
              str(self._Person__age))
        print("Gender: " +
              self._Person__gender)
```

```python
def main():
    desmond = Person("Haha", 18, "M")

    # print(f"Name: {desmond.__name}")      # Error
    # print(f"Age: {desmond.__age}")        # Error
    # print(f"Gender: {desmond.__gender}") # Error
    # Okay, but ...
    desmond.__name = "Desmond"
    # Okay, but ...
    desmond.__age = 19
    desmond.print_info()

if __name__ == "__main__":
    main()
```

Accessors and Mutators

- Accessor and mutator methods can be used to access the protected/private instance variables that cannot be accessed from outside the class.
- Accessor methods (or getters) can be used to retrieve the values of instance variables of an object. When the accessor method is called, it returns the value of the private instance variable of the object.
- Mutator methods (or setters) can be used to modify the values of instance variables of an object. When the mutator method is called, it updates the value of the private instance variable of the object.

Adding Accessors and Mutators

```
# Filename: person_w_accessors_mutators.py

class Person:
    def __init__(self, name="Tom",
                       age=18,
                       gender='M'):
        self.__name = name
        self.__age = age
        self.__gender = gender

    def getName(self):        # Accessor
        return self.__name

    def getAge(self):         # Accessor
        return self.__age

    def getGender(self):      # Accessor
        return self.__gender

    def setName(self, name): # Mutator
        self.__name = name

    def setAge(self, age):        # Mutator
        self.__age = age

    def setGender(self, gender): # Mutator
        self.__gender = gender

    def print(self):
        print("--- Print Person ---")
        print("Name: " + self.__name)
        print("Age: " + str(self.__age))
        print("Gender: " + self.__gender)

def main():
    desmond = Person("Haha", 18, "M")
    print("Name: " + desmond.getName())
    print("Age: " + str(desmond.getAge()))
    print("Gender: " + desmond.getGender())
    desmond.setName("Desmond")
    desmond.setAge(19)
    desmond.print()

if __name__ == "__main__":
    main()
```

Key Terms

- Accessor (Getter)
- Attribute
- Behavior
- Class
- Constructor
- Dot operator (.)
- Initializer
- Instance
- Instance variable
- Instance method
- Instantiation
- Mutator (Setter)
- Object-Oriented Programming (OOP)
- Private
- Protected
- Public
- Unified Modeling Language (UML)

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

1. A _____ is a template, a blueprint, a contract, and a data type for objects. It defines the properties of objects and provides an _____ for initializing objects and methods for manipulating them.
2. The initializer is always named _____ . The first parameter in each instance method including the initializer in the class refers to the object that calls the method. By convention, this parameter is named _____.
3. An _____ is an instance of a class. You use the _____ to create an object, and the _____ to access that object through the variable that references the object.

Answer: 1. class; initializer, 2. `__init__`; self, 3. object; constructor; dot operator (.).

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

4. An _____ or _____ belongs to an instance of a class. Its use is associated with individual instances.
5. You can provide a _____ method or a _____ method to enable clients to read or modify the data.

Answer: 4. instance variable; instance method, 5. accessor (getter); mutator (setter).

Further Reading

- Read Sections 9.1 - 9.8 of “Introduction to Python Programming and Data Structures” textbook.

That's all!

Any questions?